Rockpool

Release 1.0.7.post1

Dylan Muir, Felix Bauer, Philipp Weidel

Dec 03, 2019

CONTENTS

Rockpool is a Python package for working with dynamical neural network architectures, particularly for designing event-driven networks for Neuromorphic computing hardware. Rockpool provides a convenient interface for designing, training and evaluating recurrent networks, which can operate both with continuous-time dynamics and event-driven dynamics.

Rockpool is an open-source project managed by aiCTX AG.

CHAPTER

ONE

ABOUT ROCKPOOL

Rockpool is an open source project released by aiCTX AG.

1.1 About aiCTX

aiCTX is a Neuromorphic computing hardware and solutions startup, based in Zurich Switzerland. The company specializes in developing mixed-signal neuromorphic silicon hardware for neural simulation and signal processing; it develops software for interfacing with and configuring neuromorphic hardware; and develops solutions to analyse and process bio-signals. aiCTX is a commercial spin-off from the Institute of Neuroinformatics (INI), University of Zurich (UZH) and ETH Zurich (ETHZ).

1.2 About Noodle

Noodle is the mascot of Rockpool. Noodle is a Nudibranch, Glaucus marginatus. Nudibranches are a group of amazing sea snails that shed their shells after the larval stage, to display an incredible array of forms, patterns and colours. Glaucus marginatus is a species found in the Pacific ocean, and often seen at beaches and in rock pools of the eastern Australian coast.

Photograph of Noodle is CC BY 2.0 Taro Taylor

CHAPTER

TWO

INSTALLING ROCKPOOL

2.1 Base requirements

Rockpool requires Python 3.6, numpy, scipy and numba to install. These requirements will be installed by `pip` when installing Rockpool. We recommend using anaconda, miniconda or another environment manager to keep your Python dependencies clean.

2.2 Installation using `pip`

The simplest way to install Rockpool is by using `pip` to download and install the latest version from PyPI.

```
pip install rockpool
```

2.3 Dependencies

Rockpool has several dependencies for various aspects of the package. However, these dependencies are compartmentalised as much as possible. For example, Jax is required to use the Jax-backed layers (e.g. *RecRateEulerJax*); PyTorch is required to use the Torch-backed layers (e.g. *RecIAFTorch*), and so on. But if these dependencies are not available, the remainder of Rockpool is still usable.

- NEST for NEST-backed layers
- Jax for Jax-backed layers
- PyTorch for Torch-backed layers
- Brian2 for Brian-backed layers
- Matplotlib or HoloViews for plotting *TimeSeries*
- PyTest for running tests
- Sphinx, NBSphinx and Sphinx-autobuild for building documentation

To automatically install all the extra dependencies required by Rockpool, use the command

```
$ pip install rockpool[all]
```

CHAPTER

THREE

CONTRIBUTING

If you would like to contribute to Rockpool, then you should begin by forking the public repository at https://gitlab.com/ai-ctx/rockpool to your own account. Then clone your fork to your development machine

```
$ git clone https://gitlab.com/your-fork-location/rockpool.git rockpool
```

Install the package in development mode using `pip`

```
$ cd rockpool
$ pip install -e . --user
```

or

```
$ pip install -e .[all] --user
```

The main branch is `development`. You should commit your modifications to a new feature branch.

```
$ git checkout -b feature/my-feature develop
...
$ git commit -m 'This is a verbose commit message.'
```

Then push your new branch to your repository

```
$ git push -u origin feature/my-feature
```

Use the Black code formatter on your submission during your final commit. This is required for us to merge your changes. If your modifications aren't already covered by a unit test, please include a unit test with your merge request. Unit tests go in the `tests` directory.

Then when you're ready, make a merge request on gitlab.com, from the feature branch in your fork to https://gitlab.com/ai-ctx/rockpool.

3.1 Running tests

As part of the merge review process, we'll check that all the unit tests pass. You can check this yourself (and probably should before making your merge request), by running the unit tests locally.

To run all the unit tests for Rockpool, use `pytest`:

```
$ pytest tests
```

3.2 Building documentation

The Rockpool documentation requires Sphinx, NBSphinx and Sphinx-autobuild. The commands

```
$ cd docs
$ make livehtml
```

Will compile the documentation and open a web browser to the local copy of the docs.

CHAPTER

FOUR

GETTING STARTED WITH ROCKPOOL

Rockpool is designed to let you design, simulate, train and test dynamical neural networks – in contrast to standard ANNs, these networks include explicit temporal dynamics and simulation of time. Rockpool contains several types of neuron simulations, including continuous-time ("rate") models as well as event-driven spiking models. Rockpool supports several simulation back-ends, and layers with varying dynamics can be combined in the same network.

4.1 Importing Rockpool modules

[1]:
```
# - Switch off warnings
import warnings
warnings.filterwarnings('ignore')

# - Import classes to represent time series data
from rockpool import TimeSeries, TSContinuous, TSEvent

# - Import the `Network` base class
from rockpool import Network

# - Import some `Layer` classes to use
from rockpool.layers import RecRateEuler, PassThrough
```

[3]:
```
# - Import numpy
import numpy as np

# - Import the plotting library
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = [12, 4]
```

4.2 Build a small network

We will define a very small recurrent network, with one input channel, ten recurrent units, and one output channel.

[4]:
```
# - Define the network size
input_size = 1
rec_size = 10
output_size = 1

# - Define weights
```

(continues on next page)

(continued from previous page)

```
weights_in = np.random.rand(input_size, rec_size) - .5
weights_rec = np.random.randn(rec_size, rec_size) / rec_size
weights_out = np.random.rand(rec_size, output_size) - .5

# - Construct three layers
lyrInput = PassThrough(weights_in, name = "Input")
lyrRecurrent = RecRateEuler(weights_rec, name = "Recurrent")
lyrOut = PassThrough(weights_out, name = "Output")

# - Compose these into a network
net = Network(lyrInput, lyrRecurrent, lyrOut)

# - Display the network
print(net)
```

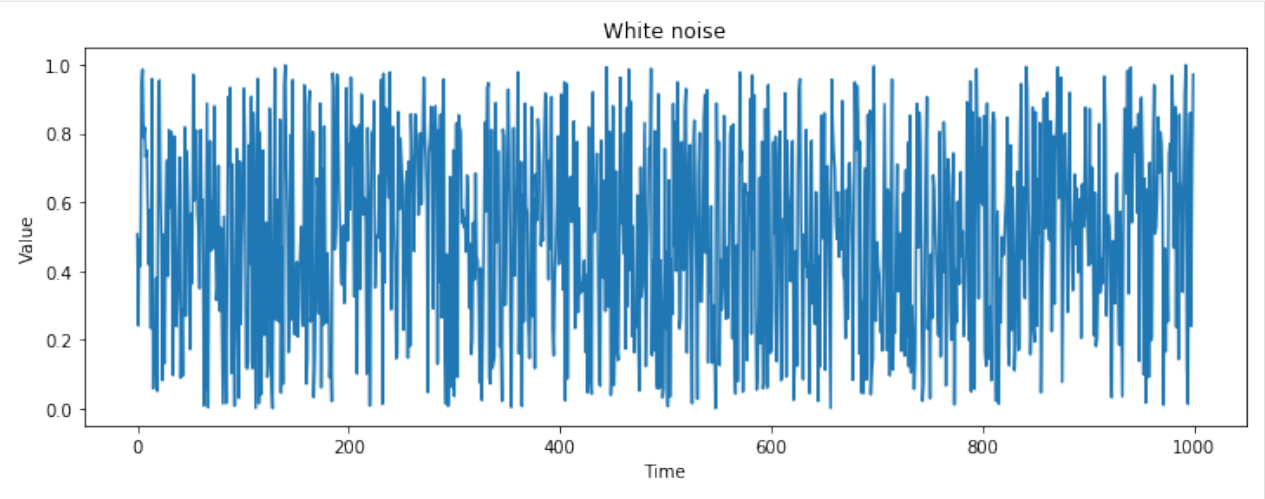
```
Network object with 3 layers
    PassThrough object: "Input" [1 TSContinuous in -> 10 internal -> 10 TSContinuous␣
↪out]
    RecRateEuler object: "Recurrent" [10 TSContinuous in -> 10 internal -> 10␣
↪TSContinuous out]
    PassThrough object: "Output" [10 TSContinuous in -> 1 internal -> 1 TSContinuous␣
↪out]
```

4.3 Define an input signal

In order to pass data through the network, we first need to generate some input data. In Rockpool, time series data is represented by *.TimeSeries* subclasses. See *Working with time series data* for more information.

Let's build a simple white noise signal as an input.

[5]:
```
# - Build a time base
timebase = np.arange(1000)

# - Create a white noise time series
ts_white = TSContinuous(timebase, np.random.rand(timebase.size))

# - Plot the time series
plt.figure()
ts_white.plot()
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('White noise');
```



4.4 Stimulate the network

Now we can inject the white noise series into the network, and obtain the resulting network activity.

[6]:
```
dResponse = net.evolve(ts_white)
```

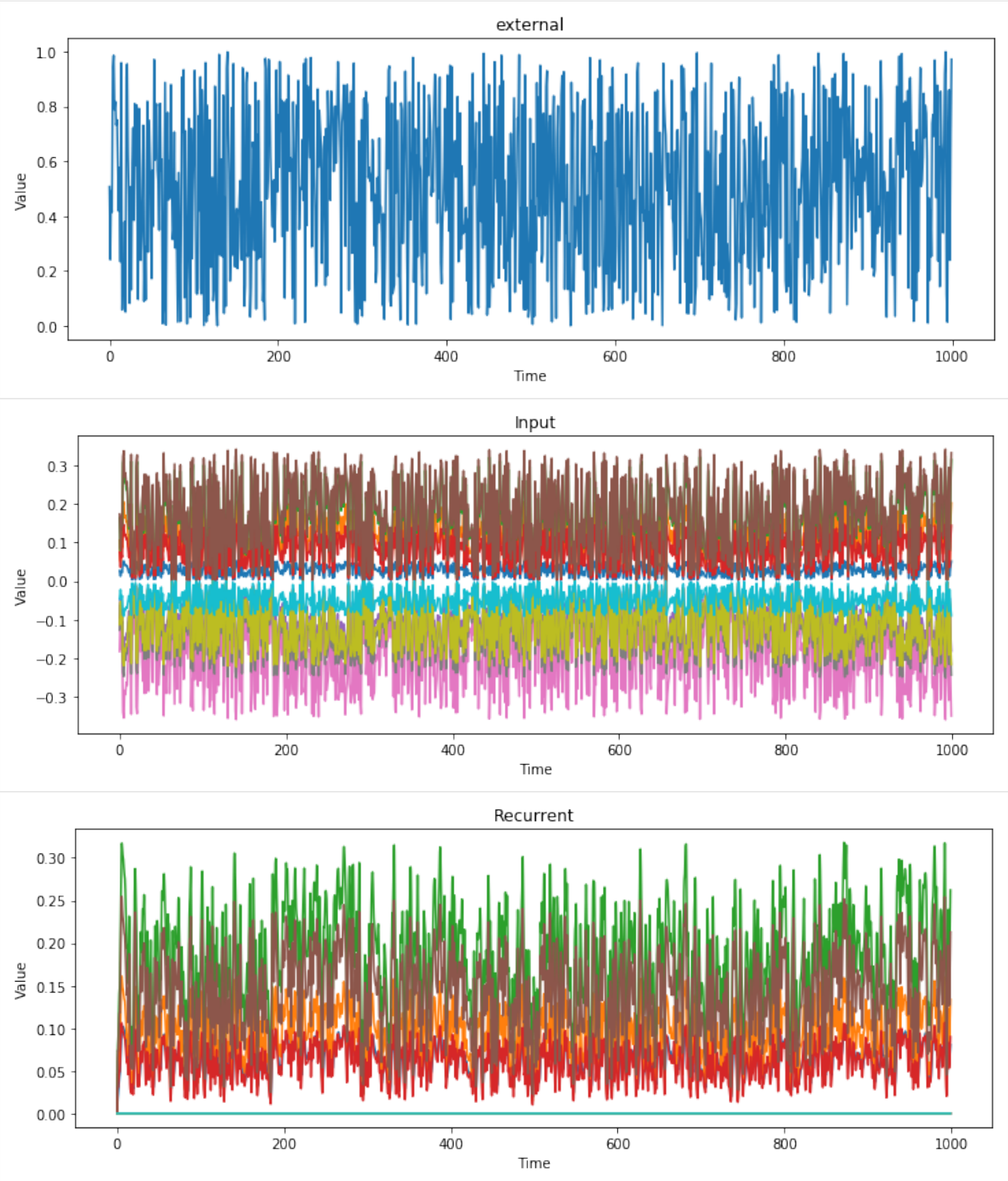
```
Network: Evolving layer `Input` with external input as input
Network: Evolving layer `Recurrent` with Input's output as input
Network: Evolving layer `Output` with Recurrent's output as input
```

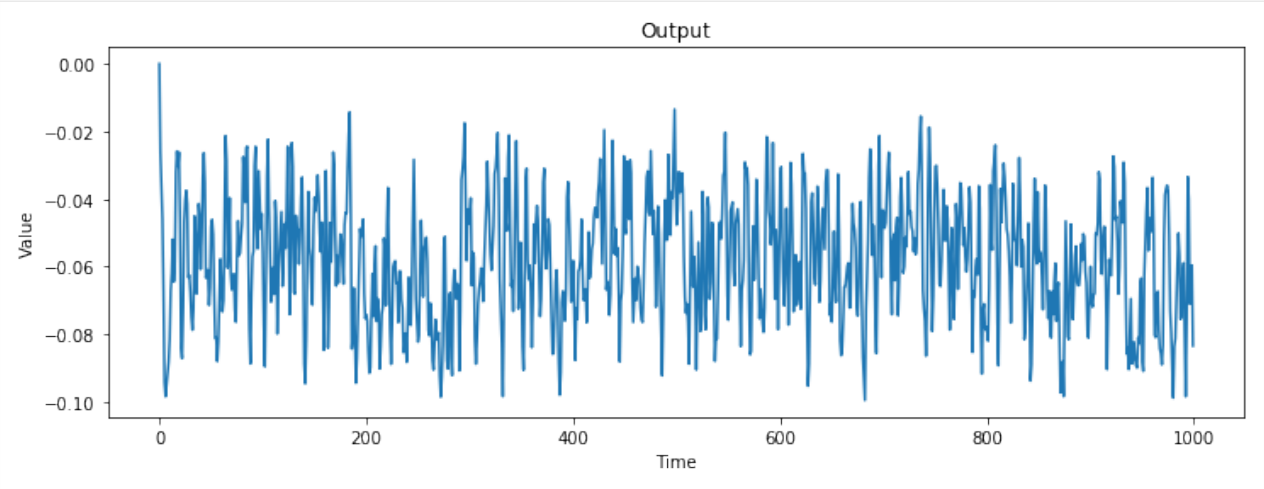
`dResponse` is a dictionary containing the time series produced by the network during evolution. The signals in `dResponse` are named for the layer which produced them:

[7]:
```
dResponse
```

[7]:
```
{'external': non-periodic TSContinuous object `unnamed` from t=0.0 to 999.0. Samples:␣
↪1000. Channels: 1,
 'Input': non-periodic TSContinuous object `unnamed` from t=0.0 to 999.0. Samples:␣
↪1000. Channels: 10,
 'Recurrent': non-periodic TSContinuous object `unnamed` from t=0.0 to 999.0. Samples:
↪ 9991. Channels: 10,
 'Output': non-periodic TSContinuous object `unnamed` from t=0.0 to 999.0. Samples:␣
↪1000. Channels: 1}
```

[8]:
```
# - Define an auxilliary function to help us with plotting
def plot_signal(k, ts):
    plt.figure()
    ts.plot()
    plt.title(k)
    plt.xlabel('Time')
    plt.ylabel('Value')

# - Plot all the signals in turn
[ plot_signal(k, ts)
    for k, ts in zip(dResponse.keys(), dResponse.values())
];
```


external
Value
Time
Input
Value
Time
Recurrent
Value
Time



Now you know how to build and simulate small networks, you can explore the other types of *.Layer* offered by Rockpool, especially event-driven layers. See *Types of Layer available in Rockpool* for more information, and :ref:'/tutorials/building_reservoir.ipynb' for a detailed view of building and training a reservoir.

Rockpool also offers facilities for training networks, and seamlessly using Neuromorphic compute hardware as simulation backends. See the tutorial on *Event-based simulation on DynapSE hardware* for more details.

CHAPTER

FIVE

WORKING WITH TIME SERIES DATA

5.1 Concepts

In Rockpool, temporal data ("time series" data) is encapsulated in a set of classes that derive from *.TimeSeries*. Time series come in two basic flavours: "continuous" time series, which have been sampled at some set of time points but which represent values that can exist at any point in time; and "event" time series, which consist of discrete event times.

The *.TimeSeries* subclasses provide methods for extracting, resampling, shifting, trimming and manipulating time series data in a convenient fashion. Since Rockpool naturally deals with temporal dynamics and temporal data, TimeSeries objects are used to pass around time series data both as input and as output.

`TimeSeries` objects have an implicit shared time-base at $t_0 = 0$ sec. However, they can easily be offset in time, concatenated, etc.

Housekeeping and import statements

[1]:
```
# - Import required modules and configure

# - Switch off warnings
import warnings
warnings.filterwarnings('ignore')

# - Required imports
import numpy as np

from rockpool.timeseries import (
    TimeSeries,
    TSContinuous,
    TSEvent,
    set_global_ts_plotting_backend,
)

# - Use HoloViews for plotting
import colorcet as cc
import holoviews as hv
hv.extension('bokeh')

%opts Curve [width=600]
%opts Scatter [width=600]
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

5.2 Continuous time series represented by `TSContinuous`

Continuous time series are represented by tuples $[t_k, a(t_k)]$, where $a(t_k)$ is the amplitude of a signal, sampled at the time t_k. A full time series is therefore the set of samples $[t_k, a(t_k)]$ for $k = 1 \ldots K$.

Continuous time series in Rockpool are represented by the *.TSContinuous* class.

A time series is constructed by providing the sample times in seconds and the corresponding sample values. The full syntax for constructing a *.TSContinuous* object is given by

```
def __init__(
    self,
    times: Optional[ArrayLike] = None,
    samples: Optional[ArrayLike] = None,
    num_channels: Optional[int] = None,
    periodic: bool = False,
    t_start: Optional[float] = None,
    t_stop: Optional[float] = None,
    name: str = "unnamed",
    interp_kind: str = "linear",
)
```

[2]:
```
# - Build a time trace vector
duration = 10.
times = np.arange(0., duration, 0.1)
theta = times / duration * 2 * np.pi

# - Create a TSContinuous object containing a sin-wave time series
ts_sin = TSContinuous(
    times=times,
    samples=np.sin(theta),
    name="sine wave",
)
ts_sin
```

[2]:
```
non-periodic TSContinuous object `sine wave` from t=0.0 to 9.9. Samples: 100.␣
↪Channels: 1
```

.TSContinuous objects provide a convenience plotting method *.TSContinuous.plot* for visualisation. This makes use of *holoviews* / *bokeh* or *matplotlib* plotting libraries, if available.

If both are available you can choose between them using the *.timeseries.set_global_plotting_backend* function.

[3]:
```
# - Set backend for Timeseries to holoviews
set_global_ts_plotting_backend("holoviews")

# # - Alternatively, it can be set for specific Timeseries instances
# ts_sin.set_plotting_backend("holoviews")

# - Plot the time series
ts_sin.plot()
```

```
Global plotting backend has been set to holoviews.
```

[3]:
```
:Curve   [Time]   (y)
```

.TSContinuous objects can represent multiple series simultaneously, as long as they share a common time base:

[4]:
```
# - Create a time series containing a sin and cos trace
ts_cos_sin = TSContinuous(
    times=times,
    samples=np.stack((np.sin(theta), np.cos(theta))).T,
    name="sine and cosine",
)
# - Print the representation
print(ts_cos_sin)

# - Plot the time series
ts_cos_sin.plot()
```

```
non-periodic TSContinuous object `sine and cosine` from t=0.0 to 9.9. Samples: 100.␣
↪Channels: 2
```

[4]:
```
:Overlay
   .Curve.I  :Curve   [Time]   (y)
   .Curve.II :Curve   [Time]   (y)
```

For convenience, *.TimeSeries* objects can be made to be periodic. This is particularly useful when simulating networks over repeated trials. To do so, use the *periodic* flag when constructing the *.TimeSeries* object:

[5]:
```
# - Create a periodic time series object
ts_sin_periodic = TSContinuous(
    times=times,
    samples=np.sin(theta),
    periodic=True,
    name="periodic sine wave",
)
# - Print the representation
print(ts_sin_periodic)

# - Plot the time series
plot_trace = np.arange(0, 100, .1)
ts_sin_periodic.plot(plot_trace)
```

```
periodic TSContinuous object `periodic sine wave` from t=0.0 to 9.9. Samples: 100.␣
↪Channels: 1
```

[5]:
```
:Curve   [Time]   (y)
```

Continuous time series permit interpolation between sampling points, using *scipy.interpolate* as a back-end. By default linear interpolation is used, but any interpolation method supported by *scipy.interpolate* can be provided as a string when constructing the *.TSContinuous* object.

The interpolation interface is simple: *.TSContinuous* objects are callable with a list-like set of time points.; the interpolated amplitudes at those time points are returned as a *numpy.ndarray*.

[6]:
```
# - Interpolate the sine wave
print(ts_sin([1, 1.1, 1.2]))
```

```
[[0.58778525]
 [0.63742399]
 [0.68454711]]
```

As a convenience, *.TSContinuous* objects can also be indexed, which uses interpolation. A second index can be provided to choose specific channels. Indexing will return a new *.TSContinuous* object.

[7]:
```
# - Slice a time series object
ts_cos_sin[:1:.09, 0].print()
```

```
non-periodic TSContinuous object `sine and cosine` from t=0.0 to 0.99. Samples: 12.␣
↪Channels: 1
0.0:     [0.]
0.09:    [0.05651147]
0.18:    [0.11282469]
0.27:    [0.16876689]
         ...
0.72:    [0.43697417]
0.8099999999999999:      [0.48716099]
0.8999999999999999:      [0.53582679]
0.99:    [0.58258941]
```

.TSContinuous provides a large number of methods for manipulating time series. For example, binary operations such as addition, multiplication etc. are supported between two time series as well as between time series and scalars. Most operations return a new *.TSContinuous* object.

See the api reference for *.TSContinuous* for full detail.

Attributes (`TSContinuous`)

Attribute name	Description
`times`	Vector T of sample times
`samples`	Matrix $T \times N$ of samples, corresponding to sample times in `times`
`num_channels`	Scalar reporting N: number of series in this object
`num_traces`	Synonym to `num_channels`
`t_start`, `t_stop`	First and last sample times, respectively
`duration`	Duration between `t_start` and `t_stop`
`plotting_backend`	Current plotting backend for this instance.

5.2.1 Examples of time series manipulation

[8]:
```
# - Perform additions, powers and subtractions of time series

(   (ts_sin + 2).plot() + \
    (ts_cos_sin ** 6).plot() + \
    (ts_sin - (ts_sin ** 3).delay(2)).plot()
).cols(1)
```

[8]:
```
:Layout
   .Curve.Sine_wave.I   :Curve   [Time]    (y)
   .Sine_and_cosine.I   :Overlay
      .Curve.I   :Curve   [Time]    (y)
      .Curve.II  :Curve   [Time]    (y)
   .Curve.Sine_wave.II  :Curve   [Time]    (y)
```

5.3 Representation of event-based time series

Sequences of events (e.g. spike trains) are represented by the *.TSEvent* class, which inherits from *.TimeSeries*.

Discrete time series are represented by tuples (t_k, c_k), where t_k are sample times as before and c_k is a "channel" associated with each sample (e.g. the source of an event).

Multiple samples at identical time points are explictly permitted such that (for example) multiple neurons could spike simultaneously.

.TSEvent objects are initialised with the syntax

```
def __init__(
    self,
    times: ArrayLike = None,
    channels: Union[int, ArrayLike] = None,
    periodic: bool = False,
    t_start: Optional[float] = None,
    t_stop: Optional[float] = None,
    name: str = None,
    num_channels: int = None,
)
```

[9]:
```
# - Build a time trace vector
times = np.sort(np.random.rand(100))
channels = np.random.randint(0, 10, (100))
ts_spikes = TSEvent(
    times = times,
    channels = channels,
)
ts_spikes
```

[9]:
```
non-periodic `TSEvent` object `None` from t=0.0003808350954009887 to 0.
↪9896131800308735. Channels: 10. Events: 100
```

[10]:
```
# - Plot the events
ts_spikes.plot()
```

[10]:
```
:Scatter   [Time]   (Channel)
```

If *.TSEvent* is called, it returns arrays of the event times and channels that fall within the defined time points and correspond to selected channels.

[11]:
```
# - Return events between t=0.5 and t=0.6
ts_spikes(.5, .6)
ts_spikes(.5, .6, channels=[3, 7, 8])
```

[11]:
```
(array([], dtype=float64), array([], dtype=int64))
```

.TSEvent also supports indexing, where indices correspond to the indices of the events in the `times` attribute. A new *.TSEvent* will be returned. For example, in order to get a new series with the first 5 events of *tsSpikes* one can do:

[12]:
```
ts_spikes[:5]
```

[12]:
```
non-periodic `TSEvent` object `None` from t=0.0003808350954009887 to 0.
↪9896131800308735. Channels: 10. Events: 5
```

.TSEvent provides several methods for combining multiple *.TSEvent* objects and for extracting data. See the API reference for *.TSEvent* for full details.

Attributes (`TSEvent`)

Attribute name	Description
`times`	Vector T of sample times
`channels`	Vector of channels corresponding to sample times in `times`
`num_channels`	Scalar reporting N: number of channels in this object
`t_start`, `t_stop`	First and last sample times, respectively
`duration`	Duration between `t_start` and `t_stop`
`plotting_backend`	Current plotting backend for this instance.

CHAPTER

SIX

BUILDING AND SIMULATING A RESERVOIR NETWORK

This tutorial illustrates how to use Rockpool to construct, simulate, train and visualise networks (with a focus on reservoir networks).

Housekeeping and import statements

[1]:
```
# - Import required modules and configure

# - Disable warning display
import warnings
warnings.filterwarnings('ignore')

# - Required imports
import numpy as np
import scipy.signal as sig

from rockpool.timeseries import (
    TimeSeries,
    TSContinuous,
    TSEvent,
    set_global_ts_plotting_backend,
)

# - Use HoloViews for plotting
import colorcet as cc
import holoviews as hv
hv.extension('bokeh')

%opts Curve [width=600]
%opts Scatter [width=600]
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

6.1 General concepts — Networks and Layers

Networks in Rockpool are represented as stacks of layers, currently with the restriction that layers are connected in a chain from input to output. Full recurrent connectivity is permitted within a layer.

.Layer objects combine a number of neurons of arbitrary type, along with a set of weights.

In the case of feedforward layers, the weights comprise an $M \times N$ matrix W, which describe the transformation between M input channels and the N neurons in the layer.

Most recurrent layers contain two sets of weights: An $M \times N$ matrix for mapping the input to the neurons, as with feedforward layers, as well as an $N \times N$ matrix for the recurrent connections. Note that some older layer classes only have the recurrent weight matrix. Here, inputs are mapped 1:1 to the neurons and need to be of the same dimension as the layer. Use feedforward layers to map between different layer dimensions.

Different layers implement different types of neurons, and define their outputs in slightly different ways. The principal difference is whether a layer expects continous-time or event-based (i.e. spiking) inputs, and similarly what output representation the layer generates.

To be connected together, two layers must match in terms of output→input dimensionality and signal representation. There are several feedforward layers that convert between spiking and continuous signals (see below).

For a full overview of available layers, see layersdocs and the API reference.

6.2 Summary of available layers

Simple layers

Class	Structure	Input representation	Output representation	Comment
`FFRateEuler`	Feedforward	Continuous	Continuous	
`PassThrough`	Feedforward	Continuous	Continuous	Simple weighting
`PassThroughEvents`	Feedforward	Spiking	Spiking	Route events between channels
`FFIAFBrian`	Feedforward	Continuous	Spiking	
`FFIAFSpkInBrian`	Feedforward	Spiking	Spiking	
`FFExpSyn`	Feedforward	Spiking	Continuous	Filter with exponential kernel
`FFExpSynBrian`	Feedforward	Spiking	Continuous	Filter with exponential kernel (Brian2-based, obsolete)
`FFCLIAF`	Feedforward	Spiking	Spiking	Constant leak, clocked
`FFUpDown`	Feedforward	Continuous	Spiking	Analog-to-spike conversion through delta modulation
`SoftMaxLayer`	Feedforward	Spiking	Continuous	SoftMax operation
`RecRateEuler`	Recurrent	Continuous	Continuous	
`RecIAFBrian`	Recurrent	Continuous	Spiking	
`RecIAFSpkInBrian`	Recurrent	Spiking	Spiking	
`RecFSSpikeEuler`	Recurrent	Continuous	Spiking	Precise spiking timing, fast/slow synapses
`RecDynapseBrian`	Recurrent	Spiking	Spiking	DynapSE simulation
`RecCLIAF`	Recurrent	Spiking	Spiking	Constant leak, clocked
`RecDIAF`	Recurrent	Spiking	Spiking	Digital neuron. Event based.
`RecRateEulerJax`	Recurrent	Continuous	Continous	Jax-backed.
`ForceRateEulerJax`	Recurrent	Continuous	Continuous	Jax-backed. For reservoir transfer.

PyTorch- and Nest-accelerated versions

Base class	PyTorch class	Nest class	Purpose
`FFExpSyn`	`FFExpSynTorch`	–	Filter with exponential kernel
`FFIAFBrian`	`FFIAFRefrTorch`	`FFIAFNest`	FF Cont -> Spiking
–	`FFIAFTorch`	–	FF Cont -> Spiking without refractoriness but faster
`FFIAFSpkInBrian`	`FFIAFSpkInRefrTorch`	–	FF Spiking -> Spiking
–	`FFIAFSpkInTorch`	–	FF Spiking -> Spiking without refractoriness but faster
`RecIAFBrian`	`RecIAFRefrTorch`	–	Rec Cont -> Spiking
–	`RecIAFTorch`	–	Rec Cont -> Spiking without refractoriness but faster
`RecIAFSpkInBrian`	`RecIAFSpkInRefrTorch`	`RecIAFSpkInNest`	Rec Spiking -> Spiking
–	`RecIAFSpkInTorch`	–	Rec Spiking -> Spiking without refractoriness but faster
–	`RecIAFSpkInRefrCLTorch`	–	Rec Spiking -> Spiking with constant leak
–	–	`RecAEIFSpkInNest`	Rec Spiking -> Spiking with AdEx neurons

Layers for simulation of or interaction with hardware

Class	Structure	Input representation	Output representation	Purpose
`VirtualDynapse`	Recurrent	Spiking	Spiking	Conceptual simulation of DynapSE and related chips. `RecAEIFSpkInNest` as backend.
`RecDynapSE`	Recurrent	Spiking	Spiking	Set up reservoirs on DynapSE chip.

6.3 Importing the packages

[2]:
```
# - Import the network module
from rockpool.networks.network import Network

# - Import single layer classes
from rockpool.layers import RecFSSpikeEulerBT as RecSpike

# - Import the TimeSeries classes
from rockpool.timeseries import TSContinuous, TSEvent
set_global_ts_plotting_backend('holoviews')
```

```
Global plotting backend has been set to holoviews.
```

6.4 Building and simulating a simple reservoir network

Let's begin by building a simple network, with a input layer (FF rate layer); a recurrent reservoir (rate); and a linear readout (passthrough). We'll combine these into a chain of layers.

First we need to load the required classes.

[3]:
```
# - Import classes
from rockpool.layers import PassThrough, FFRateEuler
from rockpool.layers import RecRateEuler

# - Import weight generation functions
from rockpool.weights import unit_lambda_net
```

Now we need to generate layers, by providing weights to be encapsulated by each layer. We also define the network size.

[4]:
```
# - Define layer sizes
nInputChannels = 1
nRecurrentUnits = 100
nOutputChannels = 2

# - Define the input layer
lyrInput = PassThrough(weights = np.random.rand(nInputChannels, nRecurrentUnits)-.5,
                       name = 'Input',
                      )
print(lyrInput)

# - Define the recurrent layer, using a convenience weight generation function
# - USe a range of time constants, to improve performance
vtTimeConstants = np.random.rand(nRecurrentUnits) * 50 + 10
lyrRecurrent = RecRateEuler(weights = unit_lambda_net(nRecurrentUnits),
                            tau = vtTimeConstants,
                            dt = 1,
                            name = 'Reservoir',
                           )
print(lyrRecurrent)

# - Define the ouput layer
lyrOutput = PassThrough(weights = np.random.rand(nRecurrentUnits, nOutputChannels)-.5,
                        name = 'Readout',
                       )
print(lyrOutput)
```

```
PassThrough object: "Input" [1 TSContinuous in -> 100 TSContinuous out]
RecRateEuler object: "Reservoir" [100 TSContinuous in -> 100 TSContinuous out]
PassThrough object: "Readout" [100 TSContinuous in -> 2 TSContinuous out]
```

We can visualise these layers by plotting the weight matrices and eigenspectra.

[5]:
```
# - Display the layer weights
hv.Raster(lyrInput.weights
         ).redim(x = 'i_{res}', y = 'c_{in}', z = 'w') +\
hv.Raster(lyrRecurrent.weights
         ).redim(x = 'i_{res}', y = 'j_{res}', z = 'w') +\
hv.Raster(lyrOutput.weights
         ).redim(x = 'c_{out}', y = 'j_{res}', z = 'w')
```

[5]:
```
:Layout
   .Raster.I   :Raster   [i_{res},c_{in}]   (w)
   .Raster.II  :Raster   [i_{res},j_{res}]   (w)
   .Raster.III :Raster   [c_{out},j_{res}]   (w)
```

[6]:
```
# - Get the recurrent layer eigenspectrum
vfEigVals = np.linalg.eigvals(lyrRecurrent.weights)
```

(continues on next page)

(continued from previous page)

```
# - Plot the eigenspectrum
vfSamples = np.linspace(0, 2*np.pi, 100)
hv.Curve((np.cos(vfSamples), np.sin(vfSamples))
        ).options(color = 'red',
                  line_dash = 'dashed',
                  width = 340) *\
hv.Scatter((np.real(vfEigVals),
            np.imag(vfEigVals)),
          ).redim(x = 'Re',
                  y = 'Im')
```

[6]:
```
:Overlay
   .Curve.I   :Curve   [x]   (y)
   .Scatter.I :Scatter   [Re]   (Im)
```

Now we compose these layers into a network, using the `Network` class initialiser. The syntax for the initialiser is:

```
def __init__(self, *layers : Layer, tDt=None):
```

[7]:
```
# - Build a network of these layers
net = Network(lyrInput, lyrRecurrent, lyrOutput)
net
```

[7]:
```
Network object with 3 layers
    PassThrough object: "Input" [1 TSContinuous in -> 100 TSContinuous out]
    RecRateEuler object: "Reservoir" [100 TSContinuous in -> 100 TSContinuous out]
    PassThrough object: "Readout" [100 TSContinuous in -> 2 TSContinuous out]
```

6.5 Representation of time

Internally, a *.Network* has a discrete representation of time. During instantiation and whenever a layer is added or removed, it tries to find the smallest time step size `Network.dt` that is a multiple of all its layers' `Layer.dt` s. For instance, if a *.Network* instance contains three layers with time steps 0.005, 0.003 and 0.006, respectively, the network's `Network.dt` will be 0.03. This makes sure that each time the *.Network.evolve* method is called, the evolution duration is a multiple of all involved layers' time step lengths and therefore all layers evolve to the same time point.

For unfortunate choices of the layer `Layer.dt` s, such as the combination of 0.007, 0.013 and 0.043, the smallest suitable `Layer.dt` for the network is rather large (3.913 in this case, which is 559 times the smallest layer time step 0.007). It may also happen for more reasonable combinations of time steps that due to numerical errors the network simply cannot find a suitable `Layer.dt` (however, due to improved handling of real values, this has become extremely rare). In these two cases the network will raise an *AssertionError*.

This can be avoided by setting the `Network.dt` parameter at instantiation, e.g.

```
net = Network(layer1, layer2, dt=0.005)
```

This forces the network's time step length to the provided value. Whenever a layer is added to the network, it will make sure that `Network.dt` is a multiple of the new layer's `Layer.dt` and raise an *AssertionError* otherwise. This also applies to the layers added at instantiation (*layer1* and *layer2* in the example above). This procedure is numerically more stable. It can also be used to set `Layer.dt` to rather large values, such as 3.913 in the example above. It also guarantees that `Network.dt` does not change over time (e.g. when new layers are added).

Let's build a simple input time series (a ramp), and simulate the network. To do that we use the *.Network.evolve* method, which handles all the signal passing within the network, and ensure that all the layers are evolved appropriately.

[8]:
```
# - Build ramp input time series
tDt = 1
tInputDuration = 100
vtTimeTrace = np.arange(0, tInputDuration+tDt, tDt)
tsRamp = TSContinuous(
    times = vtTimeTrace,
    samples = np.repeat(vtTimeTrace, nInputChannels),
    periodic = True,
    name = 'Ramp',
)
# - Plot the input ramp
tsRamp.plot()
```

[8]:
```
:Curve   [Time]   (y)
```

[9]:
```
# - Evolve the network
tSimDuration = 1000
dOutput = net.evolve(ts_input = tsRamp,
                     duration = tSimDuration,
                    )
```

```
Network: Evolving layer `Input` with external input as input
Network: Evolving layer `Reservoir` with Input's output as input
Network: Evolving layer `Readout` with Reservoir's output as input
```

[10]:
```
# - Plot the signals of the network output
(   dOutput['external'].plot(dOutput['Input'].times) +\
    dOutput['Input'].clip(channels=range(0, nRecurrentUnits, 10)).plot().redim(y = 'y_
↪{Inp}') +\
    dOutput['Reservoir'].clip(channels=range(0, nRecurrentUnits, 10)).plot().redim(y␣
↪= 'y_{Res}') +\
    dOutput['Readout'].plot().redim(y = 'y_{RO}')
).cols(1)
```

[10]:
```
:Layout
   .Curve.Ramp  :Curve   [Time]   (y)
   .Unnamed.I   :Overlay
      .Curve.I    :Curve   [Time]   (y_{Inp})
      .Curve.II   :Curve   [Time]   (y_{Inp})
      .Curve.III  :Curve   [Time]   (y_{Inp})
      .Curve.IV   :Curve   [Time]   (y_{Inp})
      .Curve.V    :Curve   [Time]   (y_{Inp})
      .Curve.VI   :Curve   [Time]   (y_{Inp})
      .Curve.VII  :Curve   [Time]   (y_{Inp})
      .Curve.VIII :Curve   [Time]   (y_{Inp})
      .Curve.IX   :Curve   [Time]   (y_{Inp})
      .Curve.X    :Curve   [Time]   (y_{Inp})
   .Unnamed.II  :Overlay
      .Curve.I    :Curve   [Time]   (y_{Res})
      .Curve.II   :Curve   [Time]   (y_{Res})
      .Curve.III  :Curve   [Time]   (y_{Res})
      .Curve.IV   :Curve   [Time]   (y_{Res})
      .Curve.V    :Curve   [Time]   (y_{Res})
      .Curve.VI   :Curve   [Time]   (y_{Res})
```

(continues on next page)

(continued from previous page)

```
      .Curve.VII  :Curve   [Time]   (y_{Res})
      .Curve.VIII :Curve   [Time]   (y_{Res})
      .Curve.IX   :Curve   [Time]   (y_{Res})
      .Curve.X    :Curve   [Time]   (y_{Res})
   .Unnamed.III :Overlay
      .Curve.I  :Curve   [Time]   (y_{RO})
      .Curve.II :Curve   [Time]   (y_{RO})
```

6.6 Training the network

Some layers support self-training, by implementing `train_XXX()` methods. Here we'll use simple ridge regression to train the linear readout layer.

First we need to generate some targets for the reservoir outputs. How about a sin and cos curve?

[11]:
```
# - Generate target signals
vtTimeTrace = np.arange(0, tInputDuration+tDt, tDt)
tsTargets = TSContinuous(vtTimeTrace,
                         np.stack((np.sin(vtTimeTrace / tInputDuration * 2 * np.pi),
                                   np.cos(vtTimeTrace / tInputDuration * 2 * np.pi)
                                  )).T,
                         periodic = True,
                         name = 'Target',
                        )
# - Plot target signals
tsTargets = tsTargets.clip(channels=range(nOutputChannels))
tsTargets.plot()
```

[11]:
```
:Overlay
   .Curve.I  :Curve   [Time]   (y)
   .Curve.II :Curve   [Time]   (y)
```

The *.Network* class provides a method *.Network.train*, which evolves the entire network over a series of input batches, then uses an auxilliary callback function *fhTraining()* to operate on the network layers.

The syntax for *.Network.train* is given by:

```
def train(
        self,
        training_fct: Callable,
        ts_input: TimeSeries = None,
        duration: float = None,
        batch_durs: float = None,
        verbose: bool = True,
        high_verbosity: bool = False,
        ):
```

The training callback function must know about the specfic network structure and composition. This is so *.Network.train* can operate without needing to assume anything about the network structure.

`fhTraining()` is called with the syntax

```
fhTraining(netObj, dtsSignals, bFirst, bFinal)
```

Here `netObj` is a reference to the network that is being trained, so that the trianing callback can access all layers and signals as necessary.

`dtsSignals` is a dictionary of signals, containing the results of evolving the network for the current batch.

`bFirst` is a boolean flag that is `True` only when `fhTraining()` is called on the first batch, to permit any initialisation steps to take place.

`bFinal` is a boolean flag that is `True` only when `fhTraining()` is called on the final batch, so it can finalise and clean up.

Pseudo-code for an example training callback would be something like

```
def fhTraining(netObj, dtsSignals, bFirst, bFinal, tsTarget):
    # - Perform initialisation
    if bFirst:
        # - Initialise the algorithm

    # - Perform some training, operating on dtsSignals and the network
    netObj.lyrOutput.train(dtsSignals['Output'], tsTarget)

    # - Finalise training
    if bFinal:
        # - Finalise the algorithm
```

.PassThrough supports ridge regression via the *.PassThrough.train_rr* method. This method has the syntax

```
def train_rr(self,
             ts_target: TimeSeries,
             ts_input: TimeSeries = None,
             regularize: float = 0,
             is_first: bool = True,
             is_final: bool = False):
```

So let's define an auxilliary function for training.

[12]:
```
# - Define a function to set up a training problem
def train_reservoir(tsTarget):
    def training_callback(netObj, dtsSignals, bFirst, bFinal):
        # - Call layer layer training function
        netObj.output_layer.train_rr(
            ts_target = tsTarget,
            ts_input = dtsSignals['Reservoir'],
            regularize = .1,
            is_first = bFirst,
            is_last = bFinal,
        )
    return training_callback

# - Get a callback function
fhTrainingCallback = train_reservoir(tsTargets)
```

We will train the network over several batches of input data. Note that we also use a "burn-in" time, to allow the internal reservoir dynamics to settle away from their transient initial state. See the figures showing the internal state evolution above, to see this settling in action.

If we do not let the internal dynamics settle, then some of the training effort will go towards fitting the output to the transient state. This leads to bad performance later on.

[13]:
```
# - Train the network
nNumBatchesTrain = 10
tBurnInTime = 1000
```

(continues on next page)

(continued from previous page)

```
net.reset_all()
net.evolve(ts_input = tsRamp, duration = tBurnInTime)
net.train(training_fct = fhTrainingCallback,
          ts_input = tsRamp,
          duration = tInputDuration,
          batch_durs = tInputDuration / nNumBatchesTrain,
          verbose = True,
         )
```

```
Network: Evolving layer `Input` with external input as input
Network: Evolving layer `Reservoir` with Input's output as input
Network: Evolving layer `Readout` with Reservoir's output as input
```

```
HBox(children=(IntProgress(value=0, description='Network training', max=10,␣
↪style=ProgressStyle(description_wi...
```

```
Network: Training successful
```

Now that training has been completed, let's evaluate the performance of the system by injecting the ramp input and comparing the output to the target signal.

[14]:
```
dtsOutpt = net.evolve(tsRamp, duration = tInputDuration * 2)
dtsOutpt['Readout'].plot() * tsTargets.plot(dtsOutpt['Readout'].times)
```

```
Network: Evolving layer `Input` with external input as input
Network: Evolving layer `Reservoir` with Input's output as input
Network: Evolving layer `Readout` with Reservoir's output as input
```

[14]:
```
:Overlay
   .Curve.I   :Curve   [Time]   (y)
   .Curve.II  :Curve   [Time]   (y)
   .Curve.III :Curve   [Time]   (y)
   .Curve.IV  :Curve   [Time]   (y)
```

The system performs well — the signal output by the reservoir closely matches the desired target signal.

6.6.1 Training in batches

In the example above training was split into 10 batches of the same duration. However, there are a few other possible ways of defining batches, using arguments to `net.train()`:

- Setting `batch_durs` to a float: Training will be split into batches of the given duration, the last one might be shorter.
- Setting `batch_durs` with a vector with the duration for each batch. If the times don't add up to the full duration of the training, the last batch(es) will be shortened accordingly or an additional batch will be added.

Instead of time you can set the batch durations by network-timesteps by setting `nums_ts_batch` as an array of integer time steps. If both `batch_durs` and `nums_ts_batch` are provided, `nums_ts_batch` will be used.

6.7 Helper functions for building weight matrices

We provide a utility package *.weights*, containing several useful functions for constructing recurrent weight matrices. These are summarised below. See also the API reference for *.weights*.

Function	Description
`UnitLambdaNet`	Build a simple unit-circle eigenspectrum recurrent matrix, by drawing weights from a Normal distribution with std. dev. $\sigma = \sqrt{N}$
`RandomEINet`	Build a reservoir with excitatory and inhibitory populations
`RndmSparseEINet`	Build a network with excitatory and inhibitory populations, and sparse recurrent connectivity
`WilsonCowanNet`	Build a network composed of randomly-connected Wilson-Cowan units
`DiscretiseWeightMatrix`	Convert a fully-connected weight matrix into a discretised version by pruning and downsampling weights
`DynapseConform`	Build a sparse network of Normally-distributed synapses, that conforms to neurmorphic hardware constraints
`IAFSparseNet`	Build a sparse network scaled nicely for IAF spiking simulations

CHAPTER

SEVEN

GRADIENT DESCENT TRAINING OF A RATE-BASED RESERVOIR MODEL

This tutorial demonstrates using Rockpool and a *Jax*-accelerated rate-based reservoir layer to perform gradient descent training of all network parameters. The result is a trained dynamic recurrent network with long memory, optimised to perform a signal generation task.

7.1 Requirements

This example requires the Rockpool package from aiCTX, as well as *jax* and its dependencies.

[2]:
```
# - Ignore warnings
import warnings
warnings.filterwarnings('ignore')

# - Imports and boilerplate
from rockpool import TimeSeries, TSContinuous
from rockpool.layers import RecRateEulerJax, H_ReLU, H_tanh
from rockpool.layers.training import add_train_output
from tqdm import tnrange
from tqdm.autonotebook import tqdm

import numpy as np
import numpy.random as npr

import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = [12, 4]
```

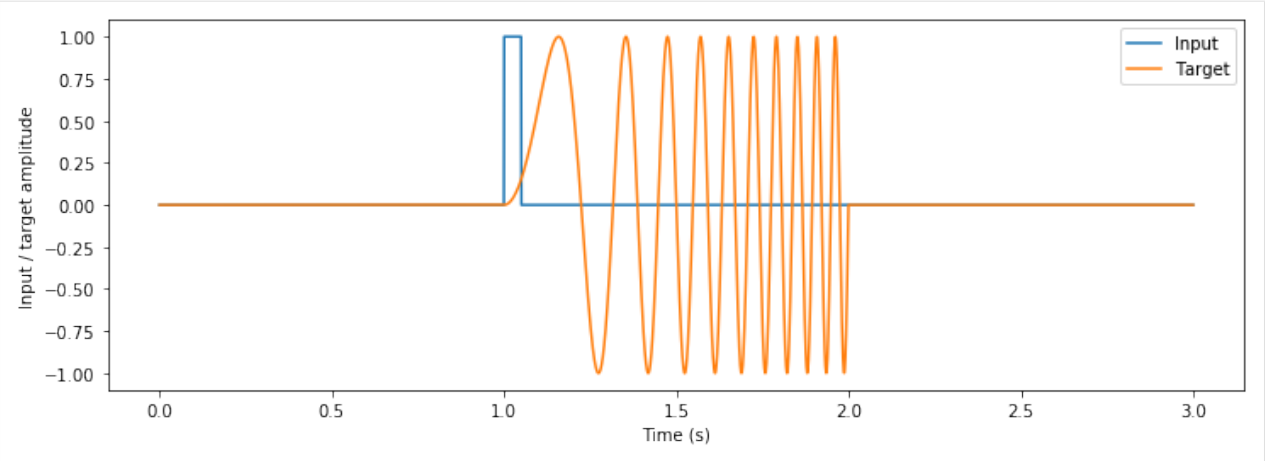
7.2 Triggered signal-generation task

We will use a pulse-to-chirp task as a demonstration. The reservoir receives a short pulse, and must respond by generating a chirp signal (a sinusoid increasing in frequency over time). This task is difficult, as no input is present for most of the time, and so considerable reservoir memory is required.

You can adjust the parameters of the input by changing the number of repeats `num_repeats`, the duration of the input pulse `pulse_duration`, and the maximum frequency reached by the chirp `chirp_freq_factor`. Shorter pulses and higher chirp frequencies make the problem more difficult. More repeats make learning more difficult, by forcing gradients to be accumulated over more time steps.

You can also adjust the time step `dt`, which makes learning slower (more time points evaluated per trial), but which permits shorter time constants to be used in the network. For numerical stability, time constants must be at least `10*dt`.

[3]:
```
# - Define input and target signals
num_repeats = 1
pulse_duration = 50e-3
chirp_freq_factor = 10
padding_duration = 1
chirp_duration = 1
dt = 1e-3

# - Build chirp and trigger signals
chirp_end = int(np.round(chirp_duration / dt))
chirp_timebase = np.linspace(0, chirp_end * dt, chirp_end)
chirp = np.atleast_2d(np.sin(chirp_timebase * 2 * np.pi * (chirp_timebase * chirp_
↪freq_factor))).T
trigger = np.atleast_2d(chirp_timebase < pulse_duration).T

# - Add padding
padding = np.zeros((int(np.round(padding_duration / dt)), 1))
chirp = np.vstack((padding, chirp, padding))
trigger = np.vstack((padding, trigger, padding))

# - Build a time base
num_samples = (chirp_end + len(padding)*2) * num_repeats
timebase = np.linspace(0, num_samples, num_samples + 1)
timebase = timebase[:-1] * dt

# - Replicate out inputs and target signals
input_t = np.tile(trigger * 1., (num_repeats, 1))
target_t = np.tile(chirp, (num_repeats, 1))

# - Generate time series objects
ts_input = TSContinuous(timebase, input_t, periodic=True)
ts_target = TSContinuous(timebase, target_t, periodic=True)

# - Plot the input and target signals
plt.figure()
plt.plot(
     timebase,
     input_t,
     timebase,
     target_t,
     # timebase, target_t + np.diff(np.vstack((target_t, target_t[0])), axis=0) / dt␣
↪* tau,
 )
plt.xlabel('Time (s)')
plt.ylabel('Input / target amplitude')
plt.legend(("Input", "Target"));
```



7.3 Network model

We will define a ReLU reservoir, with a single input and a single output channel, and with a chosen number of units in the recurrent layer `nResSize`. Larger reservoirs take longer to train, but perform the task to higher accuracy.

The dynamics of a unit j in the recurrent layer is given by

$$\tau_j \frac{\mathrm{d}x_j}{\mathrm{d}t} + x_j = W_r \cdot f(\mathbf{x}) + b_j + i_j + \sigma\zeta_j(t)$$

where τ_j is the time constant of the unit (`tau`); W_r is the $N \times N$ weight matrix defining the recurrent connections; $\mathbf{x}$ is the vector of recurrent layer activities (`w_rec`); $f(x)$ is the neuron transfer function $\tanh(x)$; b_j is the bias input of the unit (`bias`); i_j is the external input to the unit; and $\sigma\zeta_j(t)$ is a white noise process with standard deviation σ (`noise_std`).

External input is weighted such that $\mathbf{i} = W_i \cdot i(t)$, where W_i is the external input weight matrix (`w_in`) and $i(t)$ is the external input function.

The output of the reservoir is also weighted such that $z = W_o \cdot \mathbf{x}$, where W_o is the output weight matrix (`w_out`). The goal of the training task is to match the reservoir output $\hat{z}$ with a target signal z^*.

Weight initialisation doesn't seem to matter too much in this process; gradient descent does a good job even when weights are initially zero. Here we use a standard initialisation with unit spectral radius for the recurrent weights.

You can change the activation function to one of `H_tanh` or `H_ReLU`. You can also define your own, but must use `jax.numpy` to do so.

[4]:
```
# - Define the reservoir parameters
nResSize = 100
tau = 20e-3
bias = 0.
activation_func = H_ReLU
noise_std = 0.1

# - Build a layer
nInSize = 1
nOutSize = 1
w_in = 2*npr.rand(nInSize, nResSize)-1
w_rec = npr.randn(nResSize, nResSize) / np.sqrt(nResSize)
w_out = 2*npr.rand(nResSize, nOutSize)-1
```

(continues on next page)

(continued from previous page)

```
lyrRes = RecRateEulerJax(w_in, w_rec, w_out, tau, bias,
                         dt = dt, noise_std = noise_std,
                         activation_func = activation_func)

# - Get initial output
ts_output0 = lyrRes.evolve(ts_input)
```

[5]:
```
# - Initialise training
lyrRes = add_train_output(lyrRes)

# - Force initialisation of training
lyrRes.train_adam(ts_input, ts_target, is_first=True);
```

Here we use the training method `.train_adam()` to perform stochastic descent using the Adam optimiser. You can re-run the cell below as many times as you like if you want to extend the training time.

The loss function is a combination of mean-squared-error between the reservoir output and the target signal $\frac{1}{T}(\hat{z}-z^*)^2$; a factor that harshly penalises time constants τ shorter than the minimum time constant τ_{min}; and a factor related to the 2-norm of the recurrent weight matrix `np.mean(w_recurrent ** 2)` or $||W_r||^2$. This helps to ensure that the network remains stable.

At the end of each batch the network is testing by providing a pulse input, and plotting the reservoir response.

[6]:
```
num_batches = 10
trials_per_batch = 100

def two_norm(params):
    return np.sqrt(np.sum([np.sum(e ** 2) for e in params.values()]))

fig = plt.figure()
ax = plt.axes()
line_target = plt.plot(ts_target.times, ts_target.samples)
line_output = plt.plot(ts_output0.times, ts_output0.samples, '--')
plt.legend(['Target', 'Output'])
plt.xlabel('Time (s)')
plt.ylabel('Output / target');
fig.canvas.draw();

with tnrange(num_batches, desc='batches') as tqdm_batches:
    for _ in range(num_batches):
        with tnrange(trials_per_batch, desc='trials', leave=False) as tqdm_trials:
            for _ in range(trials_per_batch):
                # - Get this trial
                pass

                # - One step of Adam training
                loss, grad_loss = lyrRes.train_adam(ts_input, ts_target)

                # - Update statistics
                tqdm_trials.set_postfix(loss=loss(),
                                        #norm_g=two_norm(grad_loss()),
                                        min_tau=int(np.min(lyrRes.tau) / 1e-3),
                                        refresh=False)
                tqdm_trials.update()

            # - Update batch progress
```

(continues on next page)

(continued from previous page)

```
            tqdm_batches.update()
            ts_output = lyrRes.evolve(ts_input)
            line_output[0].set_data(ts_output.times, ts_output.samples)
            line_target[0].set_data(ts_output.times, ts_target.samples)
            ax.set_xlim([np.min(ts_output.times), np.max(ts_output.times)])
            ax.set_ylim([np.min(ts_output.samples), np.max(ts_output.samples)])
            fig.canvas.draw()
```

```
HBox(children=(IntProgress(value=0, description='batches', max=10,␣
↪style=ProgressStyle(description_width='init...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

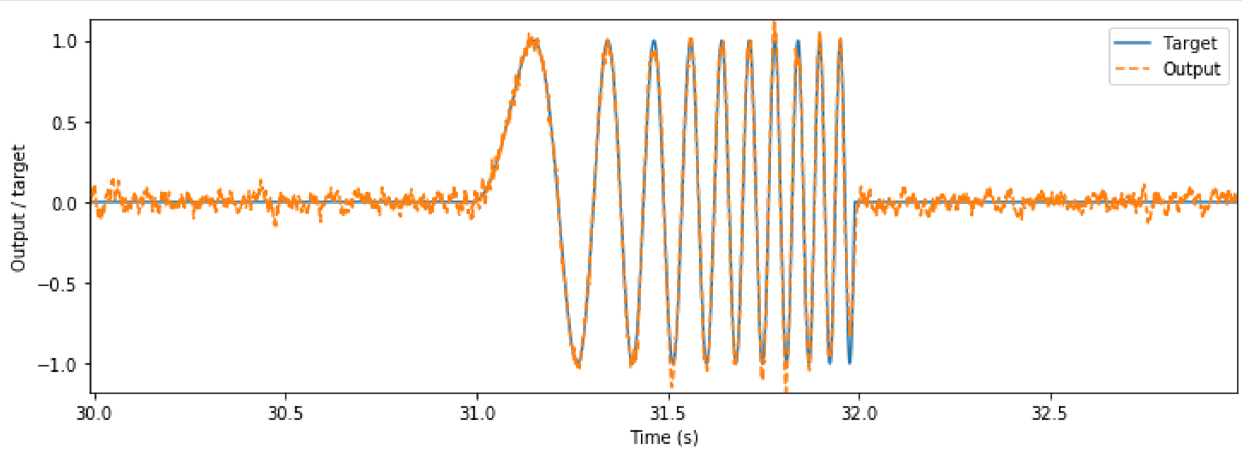
```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

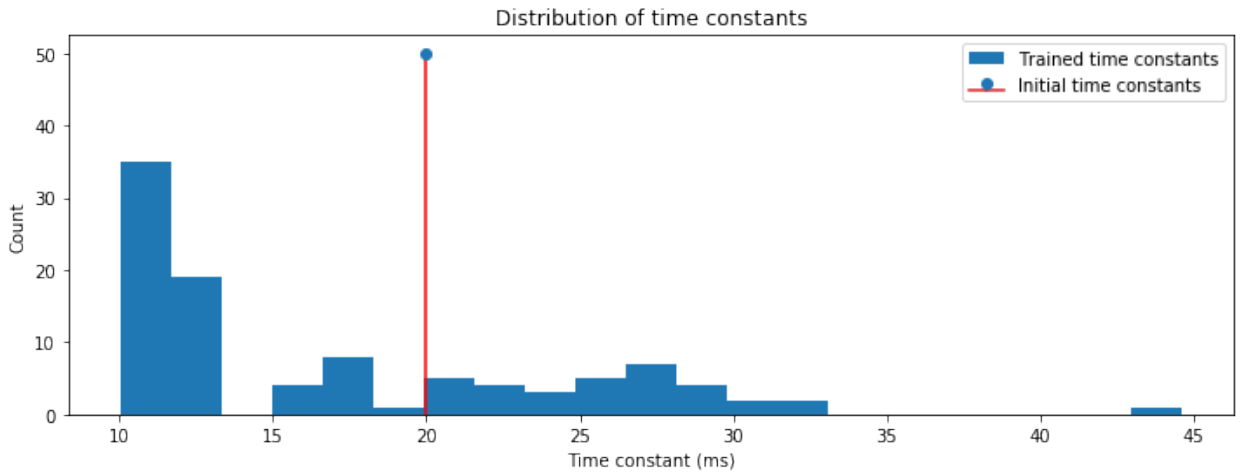
```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

```
HBox(children=(IntProgress(value=0, description='trials',␣
↪style=ProgressStyle(description_width='initial')), H...
```

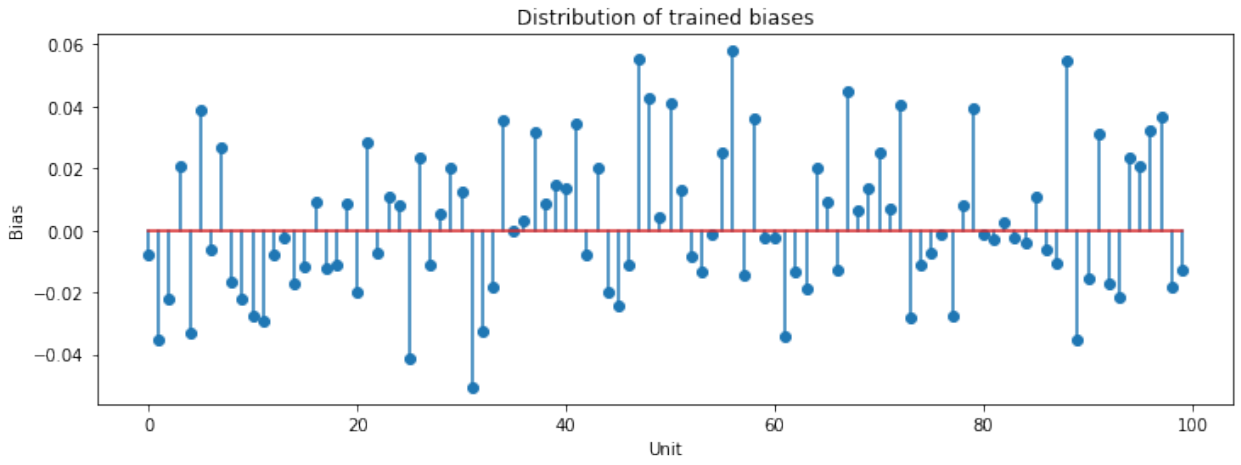


If all has gone according to plan, the output of the reservoir should closely match the target signal. We can see the effect of training by examining the distribution of network parameters below.

[9]:
```
# - Plot the network time constants
plt.figure()
plt.hist(lyrRes.tau / 1e-3, 21);
plt.stem([tau / 1e-3], [50], 'r-')
plt.legend(('Trained time constants', 'Initial time constants'))
plt.xlabel('Time constant (ms)');
plt.ylabel('Count');
plt.title('Distribution of time constants');
```



[10]:
```
# - Plot the recurrent layer biases
plt.figure()
plt.stem(lyrRes.bias);
plt.title('Distribution of trained biases')
plt.xlabel('Unit')
plt.ylabel('Bias');
```



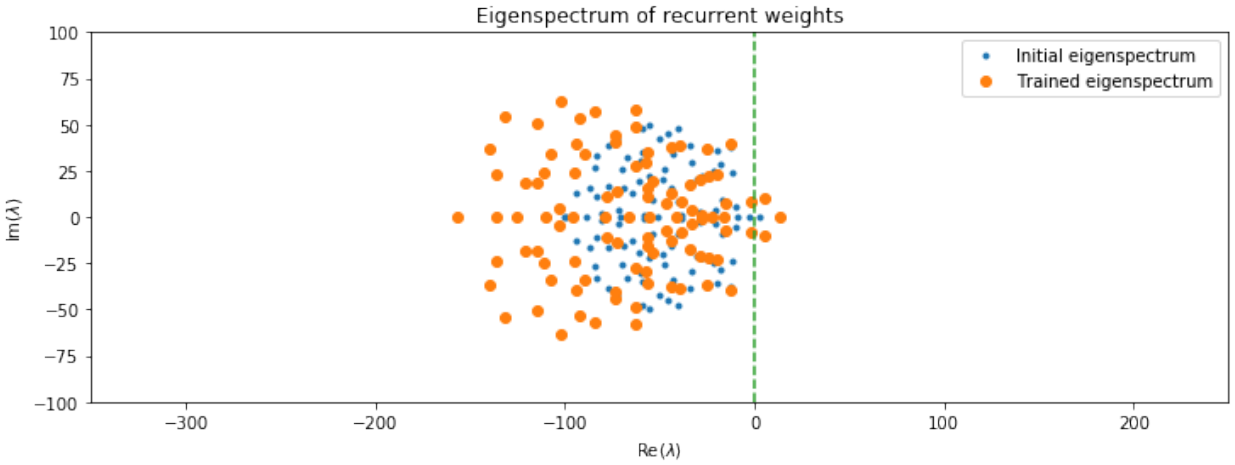
We can examine something of the computational properties of the network by finding the eigenspectrum of the Jacobian of the recurrent layer. The Jacobian $\hat{J}$ is given by

$$\hat{J} = (\hat{W}_r - I)./\hat{T}$$

where I is the identity matrix, $./$ denotes element-wise division, and T is the matrix composed of all time constants $\hat{\tau}$ of the recurrent layer.

Below we plot the eigenvalues λ of J. In my training result, several complex eigenvalues λ with real parts greater than zero are present in the trained eigenspectrum. These correspond to oscillatory modes, which are obviously useful in generating the chirp output.

[14]:
```
# - Plot the recurrent layer eigenspectrum
J = lyrRes.w_recurrent - np.identity(nResSize)
J = J / lyrRes.tau

J0 = w_rec - np.identity(nResSize)
J0 = J0 / tau

plt.figure()
eigs = np.linalg.eigvals(J)
eigs0 = np.linalg.eigvals(J0)
plt.plot(np.real(eigs0), np.imag(eigs0), '.')
plt.plot(np.real(eigs), np.imag(eigs), 'o')
plt.plot([0, 0], [-100, 100], '--')
plt.legend(('Initial eigenspectrum', 'Trained eigenspectrum',))
plt.xlim([-350, 250])
plt.ylim([-100, 100])
plt.title('Eigenspectrum of recurrent weights')
plt.xlabel('Re($\lambda$)')
plt.ylabel('Im($\lambda$)');
```



7.4 Summary

Gradient descent training does a good job of optimmising a dynamic recurrent network for a difficult task requiring significant reservoir memory. `jax` provides a computationally efficient back-end as well as automatic differentiation of the recurrent reservoir layer.

CHAPTER

EIGHT

EVENT-BASED SIMULATION ON DYNAPSE HARDWARE

This document illustrates how to use the *.RecDynapSE* layer, that runs directly on the DynapSE event-driven neuromorphic computing hardware from aiCTX.

8.1 Hardware basics

The layer uses the DynapSE processor, which consists of 4 chips. Each chips has 4 cores of 256 neurons. The chips, as well as each core in a chip and each neuron in a core are identified with an ID between 0 and 3 or 0 and 256, respectively. However, for this layer the neurons are given logical IDs from 0 to 4095 that range over all neurons. In other words the logical neuron ID is $1024 \cdot \text{ChipID} + 256 \cdot \text{CoreID} + \text{NeuronID}$.

8.2 Setup

8.2.1 Connecting to Cortexcontrol

In order to work interface the DynapSE chip, this layer relies on `cortexcontrol`. It should be accessed via an `RPyC` connection. In order to run some examples from within this jupyter notebook, we will do the latter. For this we start `cortexcontrol` and run the following commands in its console (not in this notebook):

```
[ ]: import rpyc.utils.classic
     c = rpyc.utils.classic.SlaveService()
     from rpyc.utils.server import OneShotServer
     t = OneShotServer(c, port=1300)
     print("RPyC: Ready to start.")
     t.start()
```

If the `cortexcontrol` console prints "RPyC: Ready to start." and nothing else, it is ready.

8.2.2 Using DynapseControl

The *.RecDynapSE* layer uses a *.DynapseControl* object to interact with *Cortexcontrol* (see .... for more details). You can either pass an existing *DynapseControl* object to the layer upon instantiation, or let the layer create such an object. Either way, it can be accessed via the layer's *controller* attribute.

8.3 Import

Import the class with the following command:

[2]:
```
# - Import recurrent RecDIAF layer
from rockpool.layers import RecDynapSE
```

This might take a while until the `dynapse_control` module has prepared the hardware.

8.4 Instantiation

RecDynapSE objects have many instantiation arguments in common with other Rockpool layers, in partuclar recurrent layers such as *.RecIAFTorch*, *.RecIAFBrian* or *.RecDIAF*. Other arguments are related to how the hardware is used and therefore unique to this layer.

Argument	Type	Default	Meaning
`weights_in`	2D-`ndarray`	•	Input weights (required)
`weights_rec`	2D-`ndarray`	•	Recurrent weights (required)
`neuron_ids`	1D-`ArrayLike`	`None`	IDs of the layer neurons
`virtual_neuron_ids`	1D-`ArrayLike`	`None`	IDs of the virtual (input) neurons
`dt`	`float`	`2e-5`	Time step
`max_num_trials_batch`	`int`	`None`	Maximum number of trials in individual batch
`max_batch_dur`	`float`	`None`	Maximum duration time of individual batch
`max_num_timesteps`	`int`	`None`	Maximum number of time steps in individual batch
`max_num_events_batch`	`int`	`None`	Maximum number of input Events during individual batch
`l_input_core_ids`	`ArrayLike`	`[0]`	IDs of cores that receive input spikes
`input_chip_id`	`int`	0	Chip that receives input spikes
`clearcores_list`	`ArrayLike`	`None`	IDs of cores to be reset
`controller`	`DynapseControl`	`None`	`DynapseControl` instance
`rpyc_port`	`int` or `None`	`None`	Port for RPyC connection
`name`	`str`	"unnamed"	Layer name

`weights_in` and `weights_rec` are the input and recurrent weights and have to be provided as 2D-arrays. `weights_in` determines the layer's dimensions `nSIzeIn` and `size`. `weights_rec` has to be of size `size` x `size`. Each weight must be an integer (positive or negative). Furthermore, the sum over the absolute values of the elements in any given column in `weights_in` plus the sum over the absolute values of elements in the corresponding column of `weights_rec` must be les or equal to 64. $\sum_i |$ `weights_in`$ *{ik}*| + *:nbsphinx-math:`sum`*{j}|$ `weights_rec`$ _{jk}| \leq$ `64` : $nbsphinx - math$: $forall`k$.This is due to limitations of the hardware and cannot be circumvented.$

The layer neurons of *.RecDynapSE* objects directly correspond to physical neurons on the chip. Inputs are sent to the hardware through so-called virtual neurons. Each virtual neuron has an ID, just like a physical neuron. Each input channel of the layer corresponds to such a virtual neuron.

You can choose which physical and virtual neurons are used for the layer by passing their IDs in `vnLayerNeuronIDs` and `vnVirtualNeuronIDs`, which are 1D array-like objects with integers between 0 and 1023 or 0 and 4095, respectively. All neurons with IDs from $j \cdot 256$ to $(j+1) \cdot 256,\ j \in 0, 1, ..15$ belong to the same core (with core ID $j \ mod \ 4$). All neurons with IDs from $j \cdot 1024$ to $(j+1) \cdot 1024,\ j \in 0, .., 4$ belong to same chip (with chip ID j). With this in mind you can allocate neurons to specific chips and cores.

In order for the to layer to function as expected you should stick to the following two rules: - **Neurons that receive external input should be on different cores than neurons that receive input from other layer neurons. As a consequence, each neuron should not receive both types of input.** The cores with neurons that receive external inputs are set with `l_input_core_ids`. - **All neurons that receive external input should be on the same chip.** This chip is set with `input_chip_id` and is 0 by default.

These rules are quite restrictive and it is possible to set them less strictly. Contact Felix if needed.

`.dt` is a positive `float` that on the one hand sets the discrete layer evolution timestep, as in other layers, but on the other hand also corresponds to the smallest (nonzero) interval between input events that are sent to the chip. It needs to be larger than $1.11 \cdot 10^{-9}$ (seconds). Below, under *Choosing dt* you can find some more thoughts on how to choose this value.

Evolution is automatically split into batches, the size of which is determined by `max_num_events_batch`, `max_num_timesteps`, `max_batch_dur` and `max_num_trials_batch`, which control the maximum number of events, number of timesteps, duration or number of trials in a batch. All of them can be set to `None`, which corresponds to setting no limit, except for `max_num_events_batch`, where the limit will be set to 65535. If both `max_num_timesteps` and `max_batch_dur` are not `None`, the `max_batch_dur` will be ignored. `max_num_trials_batch` will only have an effect when the input time series to the `evolve` method contains a `trial_start_times` attribute. For more details on how evolutions are split into batches see Evolution in batches.

The list `clearcores_list` contains the IDs of the cores where (presynaptic) connections should be reset on instantiation. Ideally this contains all the cores that are going to be used for this `RecDynapSE` object. However, if you want to save time and you know what you are doing you can set this to `None`, so no connections are reset.

You can pass an existing `DynapseControl` instance to the layer that will handle the interactions with `Cortexcontrol`. If this argument is `None`, a new `DynapseControl` will be instantiated. In this case, you can use the `rpyc_port` argument to define a port at which it should try to establish the connection.

.RecDynapSE.name is a *str* that defines the layer's name.

All of these values can be accessed and changed via `<Layer>.value`, where `<Layer>` is the instance of the layer.

8.4.1 Example

We can set up a simple layer on the chip by only passing input weights and recurrent weights. The weights are chosen so that there is a population of 3 "input neurons" that receive the input and then excite the remaining 6 neurons, which are recurrently connected. This way the constraints mentioned above are satisfied.

[3]:
```
import numpy as np

# - Weight matrices: 3 neurons receive external input
#   from 2 channels and stimulate the remaning
#   6 neurons, which are recurrently connected.

weights_in = np.zeros((2,9))
# Only first 3 neurons receive input
weights_in[:,:3] = np.random.randint(-2, 2, size=(2,3))

weights_rec = np.zeros((9,9))
# Excitatory connections from input neurons to rest
```

(continues on next page)

(continued from previous page)

```
weights_rec[:3, 3:] = np.random.randint(3, size=(3,6))
# Recurrent connecitons between remaining 6 neurons
weights_rec[3:, 3:] = np.random.randint(-2, 2, size=(6,6))

rlRec = RecDynapSE(weights_in, weights_rec, l_input_core_ids=[0], name="example-layer
↪")
```

```
RecDynapSE `example-layer`: Superclass initialized
dynapse_control: RPyC connection established through port 1300.
dynapse_control: RPyC namespace complete.
dynapse_control: RPyC connection has been setup successfully.
DynapseControl: Initializing DynapSE
DynapseControl: Spike generator module ready.
DynapseControl: Poisson generator module ready.
DynapseControl: Time constants of cores [] have been reset.
DynapseControl: Neurons initialized.
         0 hardware neurons and 1023 virtual neurons available.
DynapseControl: Neuron connector initialized
DynapseControl: Connectivity array initialized
DynapseControl: FPGA spike generator prepared.
DynapseControl ready.
DynapseControl: Not sufficient neurons available. Initializing  chips to make more␣
↪neurons available.
dynapse_control: Chips 0 have been cleared.
DynapseControl: 1023 hardware neurons available.
Layer `example-layer`: Layer neurons allocated
Layer `example-layer`: Virtual neurons allocated
DynapseControl: Excitatory connections of type `FAST_EXC` between virtual and␣
↪hardware neurons have been set.
DynapseControl: Inhibitory connections of type `FAST_INH` between virtual and␣
↪hardware neurons have been set.
Layer `example-layer`: Connections to virtual neurons have been set.
DynapseControl: Excitatory connections of type `FAST_EXC` between hardware neurons␣
↪have been set.
DynapseControl: Inhibitory connections of type `FAST_INH` between hardware neurons␣
↪have been set.
Layer `example-layer`: Connections from input neurons to reservoir have been set.
DynapseControl: Excitatory connections of type `SLOW_EXC` between hardware neurons␣
↪have been set.
DynapseControl: Inhibitory connections of type `FAST_INH` between hardware neurons␣
↪have been set.
DynapseControl: Connections have been written to the chip.
Layer `example-layer`: Recurrent connections have been set.
Layer `example-layer` prepared.
```

8.5 Choosing dt

As with all layers, a `RecDynapSE` object's evolution takes place in discrete time steps. This allows to send an `num_timesteps` argument to the `evolve` method, which is consistent with other layer classes and important for the use within a `Network`. Besides, event though the hardware evolves in continuous time, the input events use discrete timesteps. These timesteps are $\frac{10^{-7}}{9}$ s $= 11.\bar{1} \cdot 10^{-9}$ s and mark the smallest value that can be chosen for the layer timestep `dt`.

Although the effect the timestep size has on computation time is much smaller than with other layers, it is not always

advisable to choose the smallest possible value. The reason is that the number of timesteps between two input spikes is currently limited to $2^{16} - 1 = 65535$. This means that with dt $= 11.\overline{1} \cdot 10^{-9}$ s, any section in the input signal without input spikes longer than about 0.73 milliseconds will cause the layer to throw an exception. Therefore it makes sense to set `dt` to something between 10^{-6} and 10^{-4} seconds, in order to allow for sufficiently long silent parts in the input while still maintaining a good temporal resolution.

If these limitations are causing problems contact Felix so that he can implement a way around it.

8.6 Neurons

The layer neurons of `RecDynapSE` objects directly correspond to physical neurons on the chip. Inputs are sent to the hardware through so-called virtual neurons. Each input channel of the layer corresponds to such a virtual neuron. Every neuron on the DynapSE has a logical ID, which ranges from 0 to 4095 for the physical and from 0 to 1023 for the virtual neurons.

8.6.1 Neuron states

Hardware neurons' states change constantly according to the laws of physics, even when the layer is currently not evolving, and there is no state variable that could be read out for all neurons simultaneously. Therefore the `RecDynapSE` has no state vector like other layer classes. The `state` attribute is just a 1D array of `size` zeros.

8.7 Synapses

There are four different synapse types on the DynapSE: fast and slow excitatory as well as fast and slow inhibitory. Each neuron can receive inputs through up to 64 synapses, each of which can be any of the given types. Via `cortexcontrol` the synaptic behavior can be adjusted for each type and for each core, but not for individual synapses.

There is a priori no difference between slow and fast excitatory synapses, so they can be set to have the same behavior. In fact, one could assign shorter time constants to the "slow" excitatory synapses, making them effectively the fast ones. While both excitatory and the fast inhibitory synapses work by adding or subtracting current to the neuron membrane, the slow inhibitory synapses use shunt inhibition and in practice silence a neuron very quickly.

Note that all synapses that are of the same type and that are on the same core have the same weight. Different connection strengths between neurons can only be achieved by setting the same connection multiple times. Therefore the weight matrices `weights_in` and `weights_rec` can only be positive or negative integers and thus determine the number of excitatory and inhibitory connections to a neuron.

In this layer the connections from external input to layer neurons are fast excitatory or inhibitory synapses. Outgoing connections from neurons that receive external input are fast excitatory or inhibitory. Outgoing connecitons from other neurons are slow excitatory and fast inhibitory. For different configurations the `_compile_weights_and_configure` method has to be modified. Felix can help you with this.

8.8 Simulation

The *.evolve* method takes the standard arguments *ts_input*, *duration*, *num_timesteps* and *verbose*, which is currently not being used. Evolution duration is determined by the usual rules. If *ts_input* is provided, it must be a *.TSEvent* object.

The input is sent to the hardware neurons which will evolve and spike. The neurons' spikes are collected in a *.TSEvent* object with the timings and IDs of the spiking neurons.

Note that the hardware neurons continue evolving, so in contrast to software simulations, the layer will be in a different state (membrane potentials etc.) when an evolution begins than when the previous evolution ended. Because evolution happens in batches, this even happens between some of the timesteps within the evolution (see below).

8.8.1 Evolution in batches

As for now it is not possible to stream events (input spikes) continuously to the DynapSE. Therefore a group of events is transferred to the hardware, temporarily stored there and then translated to spikes of virtual neurons, with temporal order and inter-spike intervals matching the input signal.

The number of events that can be sent at once is limited. To allow for arbitrarily long layer evolution times, the input can be split into batches, during each of which a number of events is sent and "played back" to the hardware.

There are two ways of splitting the full input signal into batches. First, a new batch ends, as soon as it contains the maximum number of spikes or lasts the maximum allowed batch duration. These values can be set for each *.RecDynapSE* object and are stored in the *max_num_events_batch* and *max_batch_dur* attributes. While the former is limited by the hardware to be at most $2^{16}-1 = 65535$, the latter can be set to *None*, which in principle allows for arbitrarily long batch durations. However, the length of any inter-spike interval is currently limited, too, so effectively the batch duration is limited also in this case (see *Choosing dt* for more details).

Note that because the hardware neurons keep evolving after a batch ends, their state (membrane potentials etc.) will have changed until the next batch starts. These discontinuities could be problematic for some simulations. In particular if the input data consists of trials, no trial should be divided over two batches.

The second way of splitting batches considers this scenario and by splitting the input only at the beginnings of trials. The number of trials in a batch is determined by the layer's `nMaxTrialPerBatch` attribute. If a batch with this number of trials contains more events or lasts longer than allowed, the number of trials is reduced accordingly. This method is used if the input time series has a `trial_start_times` attribute, that indicates the start time of each trial, and if `max_num_trials_batch` is not `None`.

8.9 Resetting

As usual the layer's time and state can be reset by the *.RecDynapSE.reset_time*, *.RecDynapSE.reset_state* and *.RecDynapSE.reset_all* methods. However, since there is no state vector in this class, *.RecDynapSE.reset_state* has no effect and *.RecDynapSE.reset_all* effectively does the same as *.RecDynapSE.reset_time*.

8.10 Internal methods

```
_batch_input_data(
    self, ts_input: TSEvent, num_timesteps: int, verbose: bool = False
) -> (np.ndarray, int)
```

This mehtod is called by evolve, which passes it the evolution input `ts_input` and the number of evolution timesteps `num_timesteps`. It splits the input into batches according to the maximum duration, number of events and number of trials and returns a generator that for each batch yields the timesteps and channels of the input events in the batch, the time step at which the batch begins and teh duration of the batch.

```
_compile_weights_and_configure(self)
```

Configures the synaptic connections on the hardware according to the layer weights.

8.11 Class member overview

8.11.1 Methods

arguments:	Description
`_batch_input_data`	Split evolution into batches and return generator
`_compile_weights_and_configure`	Configure hardware synaptic connections
`evolve`	Evolve layer
`reset_all`	Reset layer time to 0
`reset_state`	Do nothing.
`reset_time`	Reset layer time to 0

Internal methods of parent class `Layer` *are listed in corresponding documentation.*

8.11.2 Attributes

Each argument that described in section Instantiation has a corresponding attribute that can be accessed by `<Layer>.<attribute>`, where `<Layer>` is the layer instance and `<attribute>` the argument name. Furthermore there are a few internal attributes:

Attribute name	Description
`_vHWNeurons`	1D-Array of hardware neurons used for the layer.
`vVirtualNeurons`	1D-Array of virtual neurons used for the layer.

CHAPTER

NINE

WORKING WITH SPIKING JAX LAYERS

This tutorial illustrates how to use JAX-accelerated layers in Rockpool to simulate spiking recurrent networks. The benefit of JAX backend layers is the ability to use the automatic differentiation and CPU / GPU acceleration features of JAX to perform back-propagation through time (BPTT) to train spiking recurrent networks.

[15]:
```
## Housekeeping and imports

# - Numpy
import numpy as np

# - Matplotlib and plotting config
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = [12, 4]

# - Switch off warnings
import warnings
warnings.filterwarnings('ignore')
```

9.1 Available layers

Rockpool provides three JAX-backed spiking recurrent layers, implementing leaky integrate-and-fire spiking neurons with no refractory periods. Layers with spiking input implement a single exponential synapse on the input of each layer neuron. Layers with direct current injection onto the neuron membrane are also supported.

Layer class	Description
RecLIFJax	Simple recurrent spiking layer, with spiking inputs and outputs. No I/O weighting.
RecLIFJax_IO	Recurrent spiking layer with spiking inputs, and a weighted surrogate output. Input / output weights
RecLIFCurrentInJax	Recurrent spiking layer, with direct current input injection. No I/O weighting.
RecLIFCUrrentInJax_IO	Recurrent spiking layer with direct current input injection, and a weighted surrogate output. Input /

9.2 Layer dynamics

The membrane and synaptic dynamics evolve under the equations

$$\tau_{syn}\dot{I}_{syn} + I_{syn} = 0$$

$$\tau_{mem}\dot{V}_{mem} + V_{mem} = I_{syn} + I_{in}(t) \cdot W_{in} + b + \sigma\zeta(t)$$

$$I_{syn} = I_{syn} + S_{in}(t) \cdot W_{in}$$

where I_{syn} is the N vector of synaptic input currents; V_{mem} is the N vector of membrane potentials of the neurons; b is the N vector of bias currents; $\sigma\zeta(t)$ is a white noise process with standard deviation σ, injected independently into each neuron; τ_{syn} and τ_{mem} are the N vectors of synaptic and membrane time constants, respectively; and W_{in} is the $[N_{in} \times N]$ input weight matrix.

The input spike train $S_{in}(t)$ is 1 when a spike is present on an input channel. When an input spike arrives on an input channel, the synaptic current variable is incremented by 1 then decays to zero.

The input currents $I_{in}(t)$ define input channels injeted onto neuron membranes via the input weight matrix W_{in}.

9.2.1 Spiking

When the membrane potential for neuron j, $V_{mem,j}$ exceeds the threshold voltage $V_{thr} = 0$, then the neuron j emits a spike. These events are collected into the binary spike variable $S_{rec}(t)$.

$$S_{rec}(t) = H(V_{mem} - V_{thr})$$

$$I_{syn} = I_{syn} + S_{rec}(t) \cdot W_{rec}$$

$$V_{mem} = V_{mem} - S_{rec}(t)$$

Where $H(x)$ is a Heaviside function for spike production (with a pseudo gradient described below), $H(x) = x > 0$; and W_{rec} is the matrix of recurrent synaptic weights.

All neurons therefore share a common resting potential of `0`, a firing threshold of `0`, and a subtractive reset of `-1`. The bias current for each neuron is set to `-1` by default.

The spiking events emitted by the layer $S_{rec}(t)$ are returned as the output of the

9.2.2 Surrogate signals for back-propagation

To assist with training, the layers describe here use surrogate signals in two ways: - To generate a non-spiking weighted output from V_{mem}, which can be used to train the layer to approximate a target representation - On spike generation via $H(x)$, used to propagate errors backwards through time

A surrogate $U(t)$ is generated from the layer, with $U(t) = \text{sig}\left[V_{mem}(t)\right]$, where $\text{sig}(x)$ is the sigmoid function $\text{sig}(x) = [1 + \exp(x)]^{-1}$. The surrogate weighted output of a layer is given by

$$O(t) = U(t) \cdot W_{out}$$

Where W_{out} is the $[N \times N_{out}]$ output weight matrix.

$H(x)$ is implemented internally such that it becomes differentiable in the backwards pass.

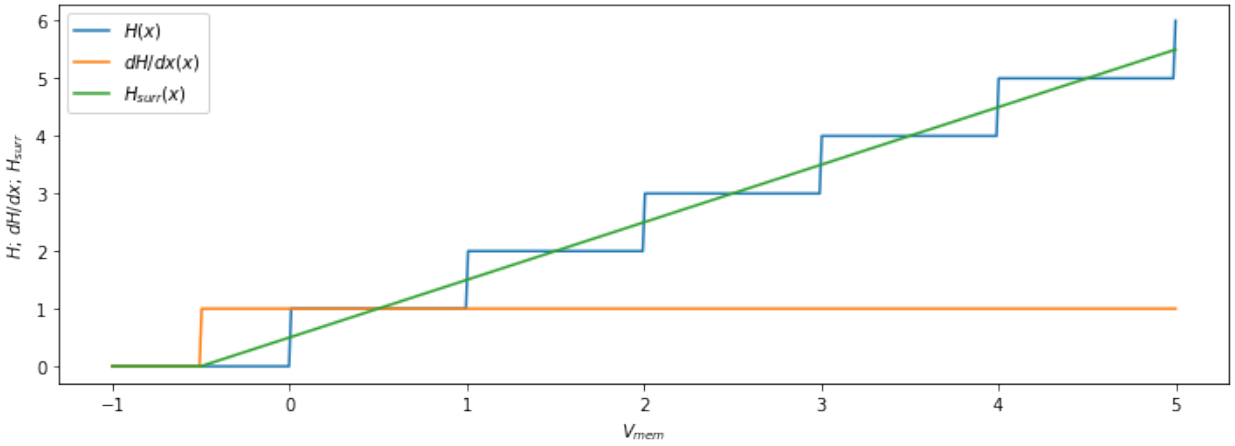
- Forward pass: $H(x) = \max\left[0, \text{floor}(x + 1)\right]$. This form permits multiple events for each neuron per time step.
- Backwards pass: $\text{d}H(x)/\text{d}x = (x > -0.5)$

[16]:
```
def H(x):
    return np.clip(np.floor(x + 1.0), 0.0, np.inf)

def dHdx(x):
    return x > -0.5

def H_surrogate(x):
    return np.clip(x+.5, 0.0, np.inf)
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-1., 5., 500)

plt.figure()
plt.plot(x, H(x), x, dHdx(x), x, H_surrogate(x))
plt.legend(['$H(x)$', '$dH/dx(x)$', '$H_{surr}(x)$'])
plt.xlabel('$V_{mem}$')
plt.ylabel('$H$; $dH/dx$; $H_{surr}$');
```



In other words, $\mathrm{d}H(x)/\mathrm{d}x(x)$ acts though each neuron is a linear-threshold unit.

9.2.3 Spiking versus current inputs

The classes *RecLIFJax* and *RecLIFJax_IO* receive spiking inputs; the *evolve()* methods for these layers accept *TSEvent* time series obejcts, describing spiking activity of the input channels over time.

The classes *RecLIFCurrentInJax* and *RecLIFCurrentInJax_IO* receive direct current injection inputs; the *evolve()* methods for these layers accept *TSContinuous* time series objects, describing instantaneous currents on the input channels.

9.2.4 Spiking versus surrogate outputs

The classes *RecLIFJax* and *RecLIFCurrentInJax* return spiking activity of the layer neurons as *TSEvent* objects from the *evolve()* method.

The classes *RecLIFJax_IO* and *RecLIFCurrentInJax_IO* return (weighted) surrogate non-spiking activity as *TSContinuous* objects from the *evolve()* method.

In both cases, several interal signals of the layers are monitored during evolution, and these are available as object attributes after evolution.

Signal	Attribute name
Spiking activity	`spikes_last_evolution`
Recurrent synaptic currents	`i_rec_last_evolution`
Surrogate signal	`surrogate_last_evolution`
Membrane voltage	`v_mem_last_evolution`
Synaptic input currents	`i_syn_last_evolution`

9.3 Building and simulating a spiking recurrent layer

We'll start by building a spiking recurrent layer *RecLIFJax*, which accepts spiking inputs and generates spiking outputs. *RecLIFJax* is imported from `rockpool.layers`.

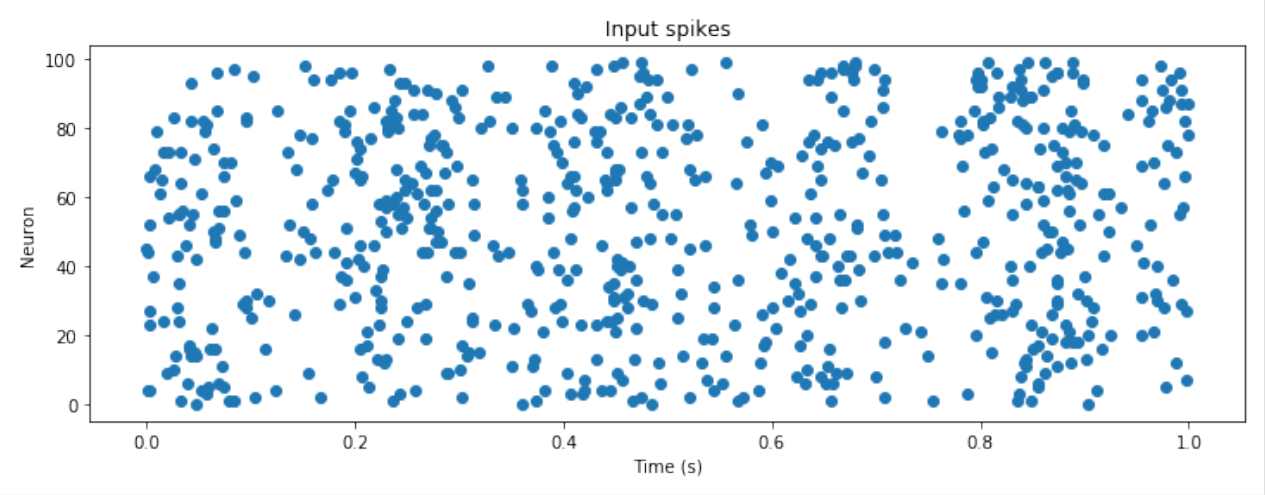
[17]:
```
## -- Import the recurrent layer
from rockpool.layers import RecLIFJax
from rockpool import TSEvent, TSContinuous
```

[18]:
```
# - Build a recurrent layer
num_neurons = 100
bias = -1
dt = 1e-3

weights_rec = 1 * np.random.randn(num_neurons, num_neurons) / np.sqrt(num_neurons)
tau_mem = np.random.rand(num_neurons) * 100e-3 + 50e-3
tau_syn = np.random.rand(num_neurons) * 100e-3 + 50e-3

lyrLIF = RecLIFJax(weights_rec, tau_mem, tau_syn, bias,
                   name = 'Recurrent LIF layer', dt = dt)
print(lyrLIF)
```

```
RecLIFJax object: "Recurrent LIF layer" [100 TSEvent in -> 100 internal -> 100␣
↪TSEvent out]
```

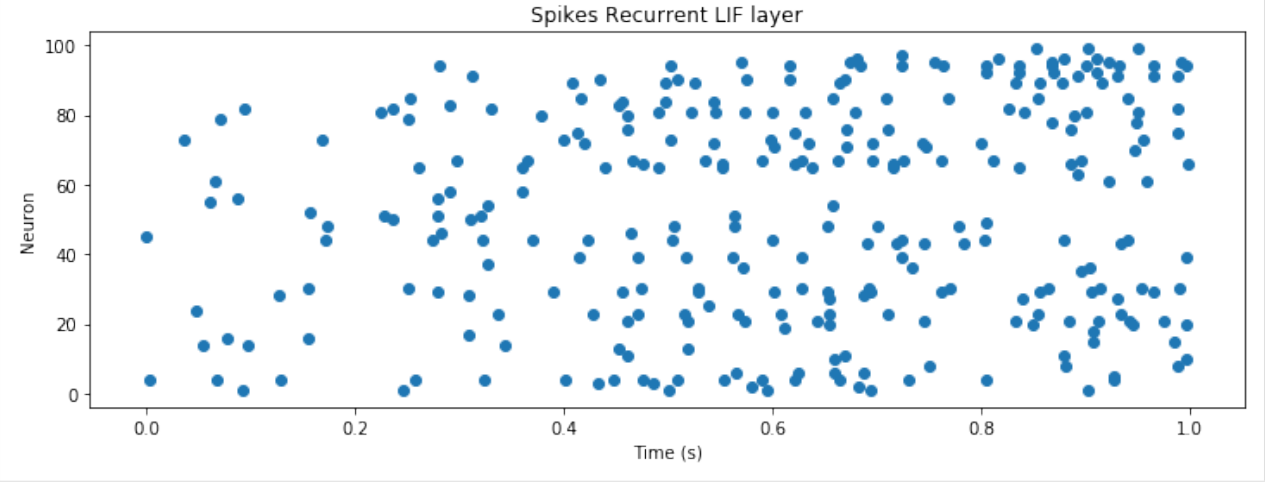
We'll generate a Poisson input signal independently for each of the layer neurons, with a sinusoidal modulation.

[19]:
```
# - Build an input signal
input_duration = 1.
mod_freq = 5.
min_rate = 3.
max_rate = 10.

# - Generate a sinusoidal instantaneous rate
time_base = np.arange(0, input_duration, dt)
sinusoid = (np.sin(time_base * 2*np.pi * mod_freq)/2. + .5) * (max_rate - min_rate) +␣
↪min_rate
sinusoid_ts = TSContinuous(time_base, sinusoid)

# - Generate Poisson spiking inputs
raster = np.random.rand(len(time_base), num_neurons) < np.tile(sinusoid * dt, (num_
↪neurons, 1)).T
spikes = np.argwhere(raster)
input_sp_ts = TSEvent(time_base[spikes[:, 0]], spikes[:, 1],
                      t_start = 0., t_stop = input_duration,
                      name = "Input spikes",
                      periodic = True,
                     )

# - Plot the input events
plt.figure()
input_sp_ts.plot()
plt.ylabel('Neuron');
```



As with all Rockpool layers, use the *evolve()* method to simulate the layer and collect the output.

[20]:
```
# - Evolve the layer with the input events
output_ts = lyrLIF.evolve(input_sp_ts)

# - Plot the output events
plt.figure()
output_ts.plot()
plt.ylabel('Neuron');
```



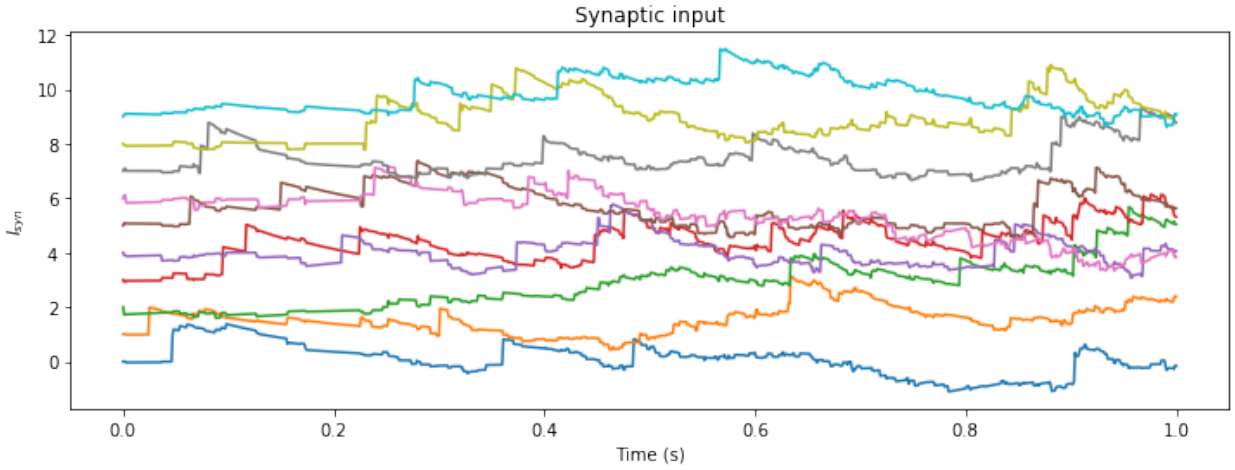
As described above, several other properties are set on each evolution, to assist in monitoring the behaviour of the layers. Let's plot those and take a look.

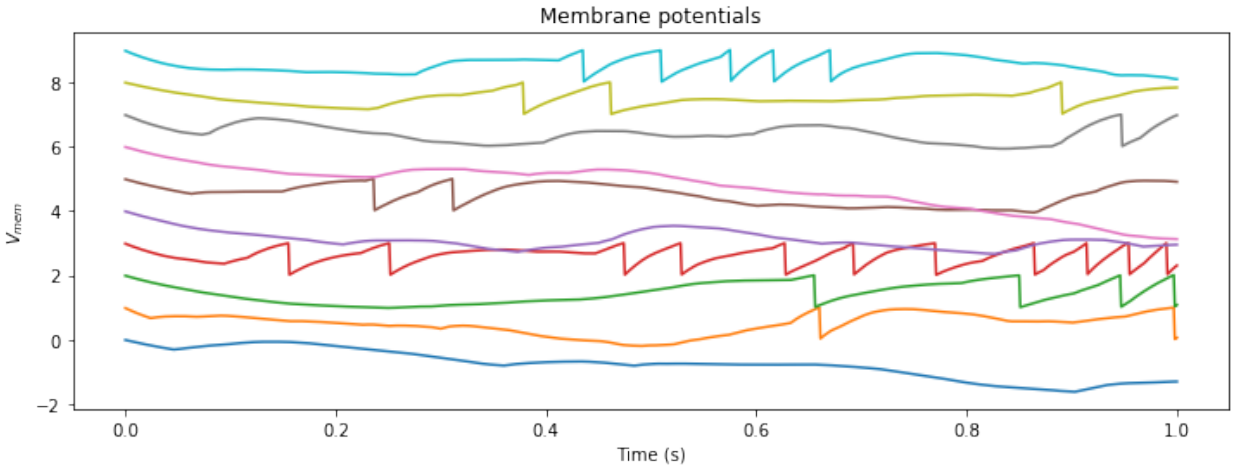
[21]:
```
# - Synaptic input currents
plt.figure()
lyrLIF.i_syn_last_evolution.plot(stagger = 1, skip = 10);
plt.xlabel('Time (s)')
plt.ylabel('$I_{syn}$')
plt.title('Synaptic input');

# - Membrane potentials
plt.figure()
lyrLIF.v_mem_last_evolution.plot(stagger = 1, skip = 10);
```

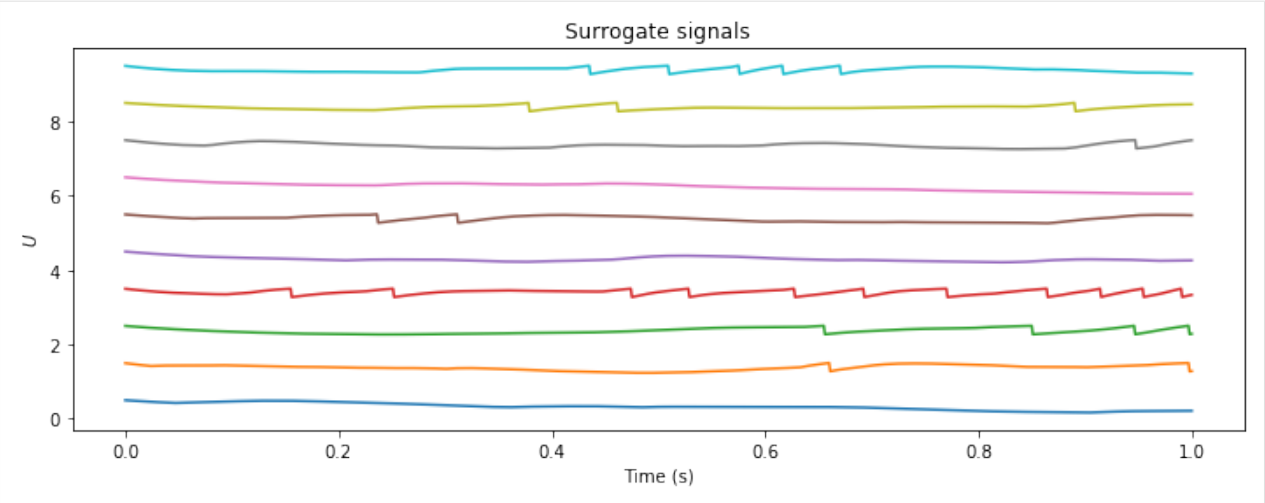
(continues on next page)

(continued from previous page)

```
plt.xlabel('Time (s)')
plt.ylabel('$V_{mem}$')
plt.title('Membrane potentials');

# - Surrogate signals
plt.figure()
lyrLIF.surrogate_last_evolution.plot(stagger = 1, skip = 10);
plt.xlabel('Time (s)')
plt.ylabel('$U$')
plt.title('Surrogate signals');
```







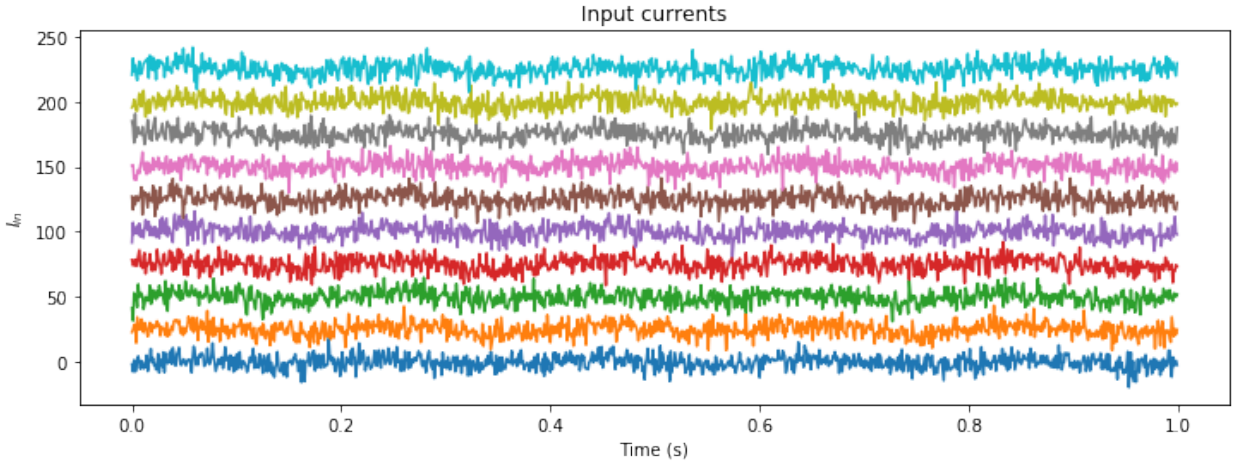
9.4 Building and stimulating a layer with current injection inputs

Interacting with a current injection layer is similar; using the *evolve()* method.

[22]:
```
# - Import a current-in class
from rockpool.layers import RecLIFCurrentInJax

# - Create a current-injection layer
lyrCI = RecLIFCurrentInJax(weights_rec, tau_mem, tau_syn,
                           name = 'Recurrent LIF layer - CI', dt = dt)
print(lyrCI)
```

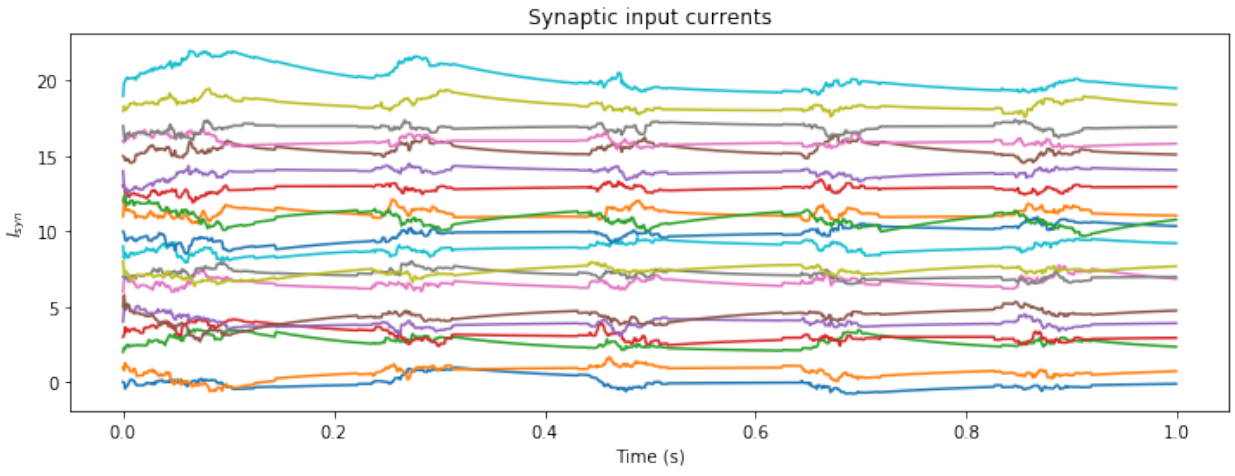
```
RecLIFCurrentInJax object: "Recurrent LIF layer - CI" [100 TSContinuous in -> 100␣
↪internal -> 100 TSEvent out]
```

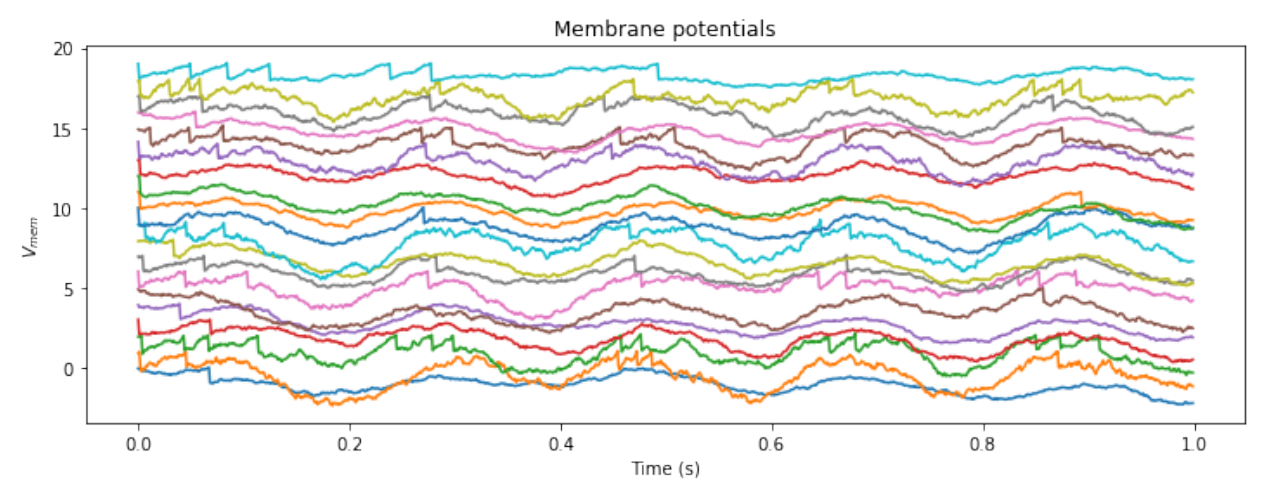
[23]:
```
# - Generate several noisy sinusoids as input currents
input_duration = 1.
mod_freq = 5.
sin_amp = 5.
noise_std = 5.

# - Generate a sinusoidal signal
time_base = np.arange(0, input_duration, dt)
sinusoid = sin_amp * np.sin(time_base * 2*np.pi * mod_freq) / 2.

inputs = np.tile(sinusoid, (num_neurons, 1)).T + noise_std * np.random.randn(len(time_
↪base), num_neurons)
input_ts = TSContinuous(time_base, inputs, periodic = True,
                        name = "Input currents",
                        units = "$I_{in}$",
                       )

# - Plot the input currents
plt.figure()
input_ts.plot(stagger = noise_std * sin_amp, skip = 10);
```

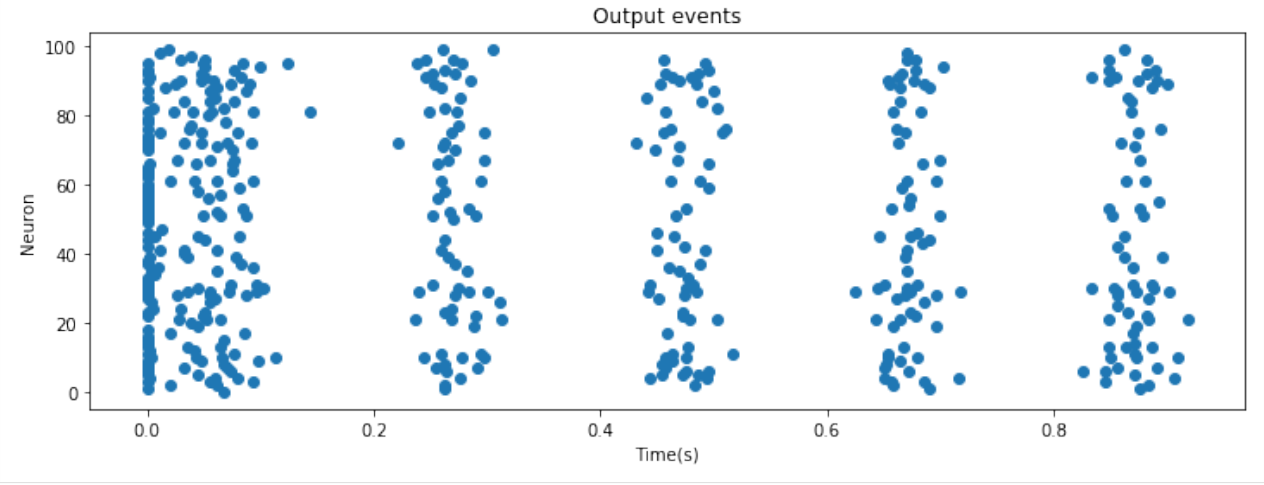


Now we can stimulate the layer using the `evolve()` method. The individual current traces will be injected directly onto the membrane of each layer neuron.

[24]:
```
# - Evolve the layer
lyrCI.evolve(input_ts)

# - Display the layer activity
plt.figure()
lyrCI.i_syn_last_evolution.plot(stagger = 1, skip = 5)
plt.ylabel('$I_{syn}$')
plt.title('Synaptic input currents')

plt.figure()
lyrCI.v_mem_last_evolution.plot(stagger = 1, skip = 5);
plt.ylabel('$V_{mem}$')
plt.title('Membrane potentials')

plt.figure()
lyrCI.spikes_last_evolution.plot();
plt.ylabel('Neuron')
plt.title('Output events')
plt.xlabel('Time(s)');
```







9.5 Using layers with input / output weighting

The layers *RecLIFJax_IO* and *RecLIFCurrentInJax_IO* accept two extra parameters, defining input and output weight matrices. Weighted outputs from these classes are formed form the neuron surrogates in both cases.

[25]:
```
# - Import a weighted layer
from rockpool.layers import RecLIFJax_IO

# - Build a layer with weighted inputs and outputs
num_in = 2
num_out = 5

w_in = np.random.rand(num_in, num_neurons)
w_rec = 1.1 * np.random.randn(num_neurons, num_neurons) / np.sqrt(num_neurons)
w_out = np.random.rand(num_neurons, num_out)

lyrIO = RecLIFJax_IO(w_in, w_rec, w_out,
                     tau_mem, tau_syn,
                     name = 'LIF with I/O weighting',
                     dt = dt,
```

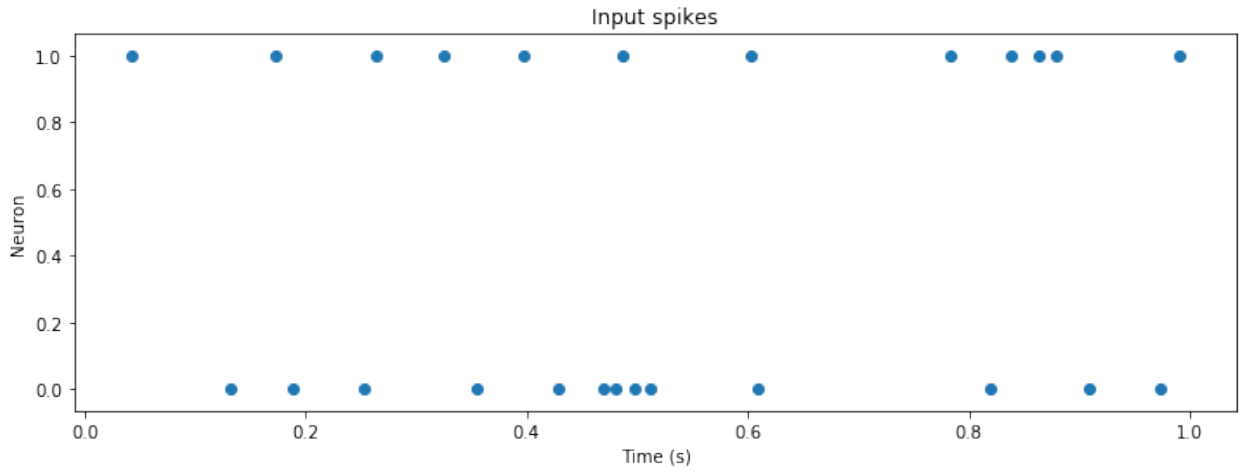
(continues on next page)

(continued from previous page)

```
                    )
print(lyrIO)
```

```
RecLIFJax_IO object: "LIF with I/O weighting" [2 TSEvent in -> 100 internal -> 5␣
↪TSContinuous out]
```

[26]:
```
# - Build an input signal
input_duration = 1.
mod_freq = 5.
min_rate = 10.
max_rate = 20.

# - Generate a sinusoidal instantaneous rate
time_base = np.arange(0, input_duration, dt)
sinusoid = (np.sin(time_base * 2*np.pi * mod_freq)/2. + .5) * (max_rate - min_rate) +␣
↪min_rate
sinusoid_ts = TSContinuous(time_base, sinusoid)

# - Generate Poisson spiking inputs
raster = np.random.rand(len(time_base), num_in) < np.tile(sinusoid * dt, (num_in, 1)).
↪T
spikes = np.argwhere(raster)
input_sp_ts = TSEvent(time_base[spikes[:, 0]], spikes[:, 1],
                      t_start = 0., t_stop = input_duration,
                      periodic = True,
                      name = "Input spikes",
                     )

# - Plot the input events
plt.figure()
input_sp_ts.plot()
plt.ylabel('Neuron');
```

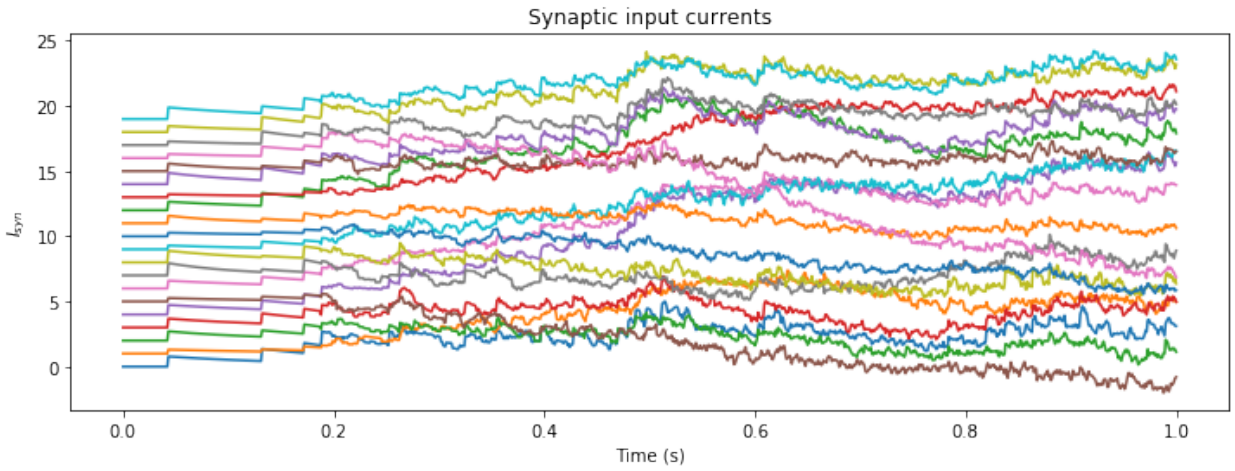


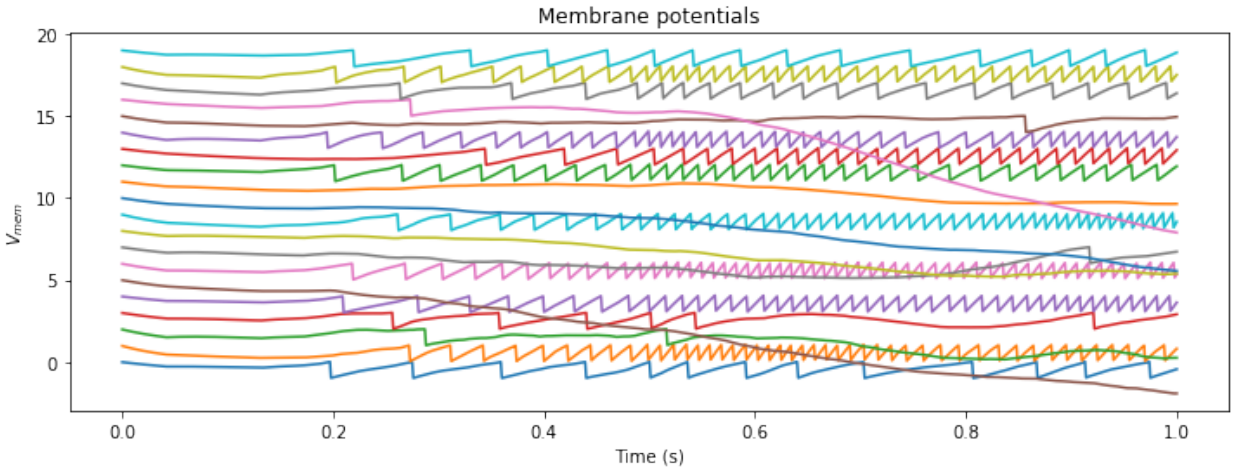
[27]:
```
# - Evolve the layer
output_ts = lyrIO.evolve(input_sp_ts)

# - Plot layer activity
plt.figure()
lyrIO.i_syn_last_evolution.plot(stagger = 1, skip = 5);
```

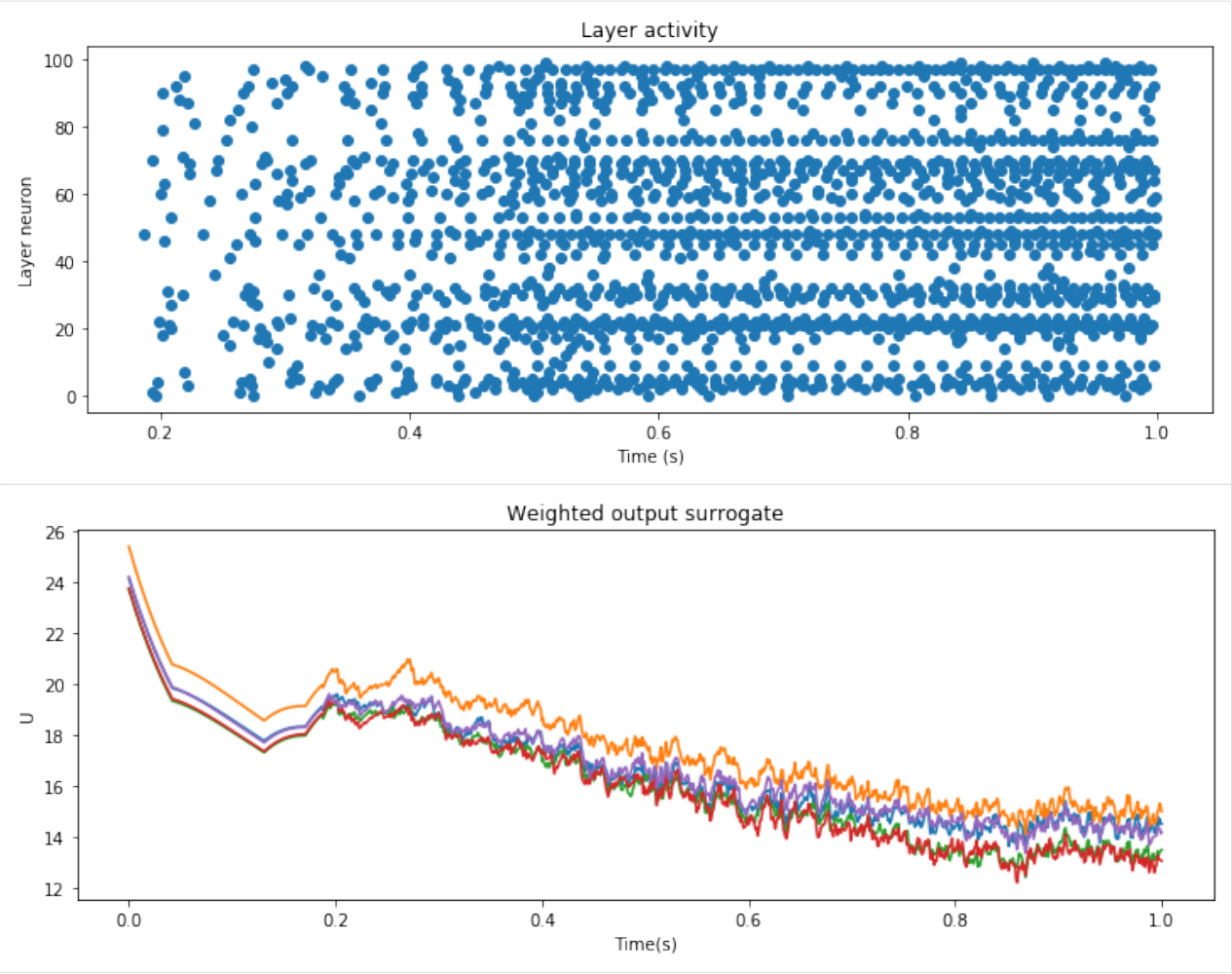
(continues on next page)

(continued from previous page)

```
plt.ylabel('$I_{syn}$')
plt.title('Synaptic input currents')

plt.figure()
lyrIO.v_mem_last_evolution.plot(stagger = 1, skip = 5);
plt.ylabel('$V_{mem}$')
plt.title('Membrane potentials')

plt.figure()
lyrIO.spikes_last_evolution.plot()
plt.ylabel('Layer neuron')
plt.title('Layer activity')

plt.figure()
output_ts.plot()
plt.xlabel('Time(s)')
plt.ylabel('U')
plt.title('Weighted output surrogate');
```







9.6 Using JAX to perform gradient-based learning

To be continued. . .

CHAPTER

TEN

WRITING A NEW `LAYER` SUBCLASS

Rockpool can be extended by writing additional *`Layer`* subclasses. This brief guide shows you how to get started.

Housekeeping and import statements

[1]:
```
# - Import required modules and configure

# - Disable warning display
import warnings
warnings.filterwarnings('ignore')

# - Required imports
import numpy as np
import scipy.signal as sig

from rockpool.timeseries import (
    TimeSeries,
    TSContinuous,
    TSEvent,
    set_global_ts_plotting_backend,
)

# - Use HoloViews for plotting
import colorcet as cc
import holoviews as hv
hv.extension('bokeh')

%opts Curve [width=600]
%opts Scatter [width=600]
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

10.1 Functionality provided by `Layer`

The *`Layer`* base class provides several attributes and methods that make implementing a new subclass easier. See also the API reference for the *`Layer`* class.

- Initialisation of attributes `weights`, `size_in`, `name`, `noise_std`, `dt`, `t`, `_time_step`

- *_prepare_input()* method, which discretises inputs to a clocked time base
- _checkinput_dims() method, which ensures that an input time series is the correct shape
- *_gen_time_trace()* method, which generates a simulation time trace
- *_expand_to_net_size()* method, which expands scalars to the dimensions of the layer, and checks that variables have the same size as the layer
- *_expand_to_weight_size()* method, which does the same as above but to the size of *.weights*
- *reset_state()* method, which sets attribute :py:attr‘~.Layer.state‘ to all zeros
- *reset_time()* method, which sets attribute :py:attr‘~.Layer._time_step‘ to zero
- *randomize_state()* method, which sets attribute :py:attr‘~.Layer.state‘ to uniformly distributed random variates
- Provides setters and getters for the attributes above
- A *reset_all()* method, which calls *reset_state()* and *reset_time()*
- Provides default attributes cInput and cOutput, which define what class of *TimeSeries* is expected for input and output

10.2 Representation of time

Each *Layer* subclass has an internal discrete representation of time. The attribute _time_step indicates how many time steps the layer object has evolved since instantiation. The length of a time step dt is normally provided as argument to the *__init__()* method. t is not an actual attribute but a property, defined as _time_step * dt. Both dt and t are in seconds.

10.3 Writing your class

Your class must inherit from *Layer* or a subclass:

```
class MyLayer(Layer):
...
```

You must provide an *evolve()* method (see below), which manages the simulation of your layer, and a *to_dict()* method, which converts the layer to a dictionary for saving.

You should probably define an *__init__()* method, which initialises everything needed for your layer (if you need to define more things). Note that the *__init__()* method does *not* call *reset_all()*, so you should do that in your *__init__()* method *after* calling super().__init__().

If your layer should accept or emit time series that are *not* *TSContinuous*, you must override the attributes cInput and cOutput appropriately. You should simply use return TSEvent or any subclass of *TimeSeries*.

10.4 Defining the evolve() method

In your subclass you must provide a method *evolve()*, which simulates the activity of neurons in the layer, and generates output signals.

The signature of this method should look like this:

```
def evolve(
    self,
    ts_input: Optional[TimeSeries] = None,
    duration: Optional[float] = None,
    num_timesteps: Optional[int] = None,
    verbose: Optional[bool] = False,
) -> TimeSeries:
    """
    Evolve the state of this layer given an input

    :param Optional[TimeSeries] ts_input: Input time series
    :param Optional[float] duration:      Simulation/Evolution time. If not provided,␣
↪the duration of `ts_input` will be used
    :param Optional[int] time_steps:      Number of evolution time steps
    :param Optional[bool] verbose:        Output information about evolution status

    :return TimeSeries: Output time series

    """
```

You can include further arguments but should provide default values in that case.

If `time_steps` is provided to the method, this is the number of time steps over which the layer has to evolve. Otherwise, this number needs to be infered from `duration` by rounding down `duration \ dt` to an integer or, if `duration` is `None`, then the duration should be inferred from *ts_input*. In this case `duration` is `ts_input.duration` if `ts_input.periodic` is `True`, or otherwise `ts_input.t_stop - self.t`. `time_steps` is then determined the same way as when `duration` is provided.

Particularly for layers with continuous-time representations, the method *`Layer._prepare_input()`* is useful:

```
def _prepare_input(self,
                   ts_input: TimeSeries = None,
                   duration: float = None,
                   num_timesteps: int = None,
) -> (np.ndarray, np.ndarray, float):
"""
_prepare_input - Sample input, set up time base

:param TimeSeries tsInput: TxM or Tx1 Input signals for this layer
:param float duration:     Duration of the desired evolution, in seconds
:param int num_timesteps:  Number of evolution time steps

:return: (time_base, input_steps, num_timesteps)
    time_base:      ndarray T1 Discretised time base for evolution
    input_steps:    ndarray (T1xN) Discretised input signal for layer
    num_timesteps:  int Actual number of evolution time steps
"""
```

This method returns a clocked time base to use for the simulation `time_base`; a discretised version of the input signals `input_steps`, clocked to the same time base; and `num_timesteps`, the actual number of time steps by which the layer should be evolved. If `num_timesteps` is provided as input argument, the returned `num_timesteps` is identical, otherwise it is inferred from `duration` or from `ts_input`. It is equal to `time_base.size - 1` because `time_base` also includes the current time point of the layer, which is not counted in `num_timesteps`.

Noise is generated within the evolve method. How you do so and how to add this to the internal state is up to you, but the attribute `fNoiseStd` is defined as the expected std. dev. of the noise after 1s of integration.

You should then evolve the activity of your layer; update `state` to be the layer state at the last time step of evolution; update `_timestep` to to reflect the new layer time, and return the output of the layer.

The precise definition of the layer output is of course up to you, but you must return a *TimeSeries*-subclass (the same class as `cOutput`). Furthermore, the output time series must cover the time range from the layer time before evolution until the end of the evolution.

10.5 Layer Properties and PyTorch backends

Many layer classes use properties with getters and setters for some attribues, instead of providing direct access to the attributes. This allows restrictions on how they are set as well as processing the data before updating the actual objects. In the following example a layer has one time constant for each neuron. The values can be accessed through an property `tau`. They are stored in an attribute `_tau`, while `tau` is only a property. This way it can be ensured that the time constants are always stored in a vector of the size of the layer.

[2]:
```
from rockpool.layers import Layer

class ExampleLayer(Layer):
    def __init__(mfW, vtTau):
        super().__init__(mfW)
        self.vtTau = vtTau

    def evolve(self):
        NotImplemented

    @property
    def vtTau(self):
        return self._vtTau

    @vtTau.setter
    def vtTau(self, vtNewTau):
        # Make sure _vtTau has correct dimensions
        self._vtTau = self._expand_to_net_size(vtNewTau, "vtTau")
```

The method *_expand_to_net_size()* is inherited from the *Layer* base class. It makes sure that the argument gets reshaped, so that it matches the size of the layer or throws an exception if this is not possible.

For layers that use `torch` as a backend, it is often helpful, if the properties return a numpy array instead of a torch tensor, so that other layers can work with it. This can be accomplished the following way:

[3]:
```
import torch

class ExampleTorchLayer(Layer):
    def __init__(self, mfW, vtTau, device):
        super().__init__(mfW)
        # Device on which torch tensors are processed
        self.device = device
        self.vtTau = vtTau

    def evolve(self):
        NotImplemented

    def to_dict(self):
        NotImplemented

    @property
    def vtTau(self):
        return self._vtTau.cpu().numpy()
```

(continues on next page)

(continued from previous page)

```
    @vtTau.setter
    def vtTau(self, vtNewTau):
        # Make sure _vtTau has correct dimensions
        _vtTau = self._expand_to_net_size(vtNewTau, "vtTau")
        # Convert to torch tensor and move to self.device
        self._vtTau = torch.from_numpy(_vtTau).float().to(self.device)
```

Here the property formalism takes care of the convesion between numpy arrays and float tensors. It also makes sure that `_tau` is on the device (e.g. cuda) that is specified in `device`.

The problem with this method is, that if we try to change the time constants via item assignment, it might behave in an unexpected way:

[4]:
```
import numpy as np
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

mfW = np.random.randn(3,4)
vtTau = np.ones(4) * 0.5
ffExample = ExampleTorchLayer(mfW, vtTau, device)

# Print the initial value
print("Initial value:", ffExample.vtTau)

# Try to assign new value to all time constants
ffExample.vtTau = 0.1
print("Changing by reassignment:", ffExample.vtTau)

# Try to assign new value to first time constant
ffExample.vtTau[0] = 1
print("Changing by item assignment:", ffExample.vtTau)
```

```
Initial value: [0.5 0.5 0.5 0.5]
Changing by reassignment: [0.1 0.1 0.1 0.1]
Changing by item assignment: [1.  0.1 0.1 0.1]
```

The time constant of the layer has not changed because the property returned a copy of `_tau` and only this copy has been modified.

This can be fixed by replacing `property` with `RefProperty`, a subclass of `property` that is defined in the `utilities` module. The object that it returns is still a copy, but with a reference to the original array which allows to change the values even with item assignment:

[6]:
```
from rockpool.utilities import RefProperty

class ExampleTorchLayer(Layer):
    def __init__(self, mfW, vtTau, device):
        super().__init__(mfW)
        # Device on which torch tensors are processed
        self.device = device
        self.vtTau = vtTau

    def evolve(self):
        NotImplemented
```

(continues on next page)

(continued from previous page)

```
    def to_dict(self):
        NotImplemented

    @RefProperty
    def vtTau(self):
        return self._vtTau

    @vtTau.setter
    def vtTau(self, vtNewTau):
        # Make sure _vtTau has correct dimensions
        _vtTau = self._expand_to_net_size(vtNewTau, "vtTau")
        # Convert to torch tensor and move to self.device
        self._vtTau = torch.from_numpy(_vtTau).float().to(self.device)


ffExample = ExampleTorchLayer(mfW, vtTau, device)

# - Print initial value
print("Initial value:", ffExample.vtTau)

# Try to assign new value to all time constants
ffExample.vtTau = 0.1
print("Changing by reassignment:", ffExample.vtTau)

# Try to assign new value to first time constant
ffExample.vtTau[0] = 1
print("Changing by item assignment:", ffExample.vtTau)
```

```
Initial value: [0.5 0.5 0.5 0.5]
Changing by reassignment: [0.1 0.1 0.1 0.1]
Changing by item assignment: [1.  0.1 0.1 0.1]
```

Note that now the function of in property only returns `_tau` instead of `Layer._tau.cpu().numpy()`. This is important because the `RefProperty` needs a reference to the actual object that has to bee modified. It will take care of the conversion between torch tensors and other array-like objects.

CHAPTER

ELEVEN

TYPES OF `LAYER` AVAILABLE IN ROCKPOOL

11.1 Rate-based non-spiking layers

layers.PassThrough(weights[, dt, noise_std, ...])	Feed-forward layer with neuron states directly corresponding to input with an optional delay
layers.FFRateEuler(weights[, dt, name, ...])	Feedforward layer consisting of rate-based neurons
layers.RecRateEuler(weights[, bias, tau, ...])	A standard recurrent non-spiking layer of dynamical neurons

11.1.1 JAX-based backend

layers.RecRateEulerJax(w_in, w_recurrent, ...)	`JAX`-backed firing-rate recurrent layer
layers.ForceRateEulerJax(w_in, w_out, tau, bias)	Implements a pseudo recurrent reservoir, for use in reservoir transfer

11.2 Event-driven spiking layers

layers.PassThroughEvents(weights[, dt, ...])	Pass through events by routing to different channels
layers.FFExpSyn(weights[, bias, dt, ...])	Define an exponential synapse layer with spiking inputs and current outputs
layers.RecDIAF(weights_in, weights_rec[, ...])	Define a spiking recurrent layer based on quantized digital IAF neurons
layers.RecFSSpikeEulerBT([weights_fast, ...])	Implement a spiking reservoir with tight E/I balance.
layers.FFUpDown(weights[, repeat_output, ...])	Define a spiking feedforward layer to convert analogue inputs to up and down channels

11.2.1 JAX-based backend

layers.RecLIFJax(w_recurrent, tau_mem, tau_syn)	Recurrent spiking neuron layer (LIF), spiking input and spiking output.
layers.RecLIFCurrentInJax(w_recurrent, ...)	Recurrent spiking neuron layer (LIF), current injection input and spiking output.

Continued on next page

Table 4 – continued from previous page

layers.RecLIFJax_IO(w_in, w_recurrent, ...)	Recurrent spiking neuron layer (LIF), spiking input and weighted surrogate output.
layers.RecLIFCurrentInJax_IO(w_in, ...[, ...])	Recurrent spiking neuron layer (LIF), weighted current input and weighted surrogate output.

11.2.2 Layers with constant leak

layers.CLIAF(weights_in[, bias, v_thresh, ...])	Abstract layer class of integrate and fire neurons with constant leak
layers.FFCLIAF(weights[, bias, v_thresh, ...])	Feedforward layer of integrate and fire neurons with constant leak
layers.RecCLIAF(weights_in, weights_rec[, ...])	Recurrent layer of integrate and fire neurons with constant leak
layers.SoftMaxLayer([weights, thresh, dt, name])	A spiking SoftMax layer with spiking inputs and outputs, and constant leak
layers.FFCLIAFCNNTorch	

11.2.3 Brian-based backend

layers.FFIAFBrian(weights[, bias, dt, ...])	*DEPRECATED* A spiking feedforward layer with current inputs and spiking outputs
layers.FFIAFSpkInBrian(weights[, bias, dt, ...])	*DEPRECATED* Spiking feedforward layer with spiking inputs and outputs
layers.RecIAFBrian([weights, bias, dt, ...])	*DEPRECATED* A spiking recurrent layer with current inputs and spiking outputs, using a Brian2 backend
layers.RecIAFSpkInBrian(weights_in, weights_rec)	*DEPRECATED* Spiking recurrent layer with spiking in- and outputs, and a Brian2 backend
layers.FFExpSynBrian([weights, dt, ...])	*DEPRECATED* Define an exponential synapse layer (spiking input), with a Brian2 backend

11.2.4 Torch-based backend

layers.FFExpSynTorch([weights, bias, dt, ...])	Define an exponential synapse layer (spiking input, pytorch as backend)
layers.FFIAFTorch(weights[, bias, dt, ...])	Define a spiking feedforward layer with spiking outputs, with a PyTorch backend
layers.FFIAFRefrTorch(weights[, bias, dt, ...])	A spiking feedforward layer with spiking outputs and refractoriness
layers.FFIAFSpkInTorch(weights[, bias, dt, ...])	Spiking feedforward layer with spiking in- and outputs
layers.FFIAFSpkInRefrTorch(weights[, bias, ...])	Spiking feedforward layer with spiking in- and outputs and refractoriness, using a PyTorch backend
layers.RecIAFTorch(weights[, bias, dt, ...])	A spiking recurrent layer with input currents and spiking outputs
layers.RecIAFRefrTorch(weights[, bias, dt, ...])	A spiking recurrent layer with current inputs, spiking outputs and refractoriness.
layers.RecIAFSpkInTorch(weights_in, weights_rec)	A spiking recurrent layer with spiking in- and outputs

Continued on next page

Table 7 – continued from previous page

layers.RecIAFSpkInRefrTorch(weights_in, ...)	A spiking recurrent layer with spiking in- and outputs and refractoriness, and a PyTorch backend
layers.RecIAFSpkInRefrCLTorch(weights_in, ...)	A recurrent spiking layer with constant leak.
layers.FFCLIAFCNNTorch	

11.2.5 Nest-based backend

layers.FFIAFNest	
layers.RecIAFSpkInNest	
layers.RecAEIFSpkInNest	

11.2.6 Hardware-backed and hardware simulation

For more information on using these layers, see *Event-based simulation on DynapSE hardware*

layers.RecDynapSE(weights_in, weights_rec[, ...])	Recurrent spiking layer implemented with a DynapSE backend.
layers.VirtualDynapse	

CHAPTER

TWELVE

FULL API SUMMARY FOR ROCKPOOL

12.1 Base classes

See also:

Getting started with Rockpool and *Working with time series data*.

networks.Network(*layers[, dt])	Base class to manage networks (collections of *Layer* objects)
layers.Layer(weights[, dt, noise_std, name])	Base class for Layers in rockpool

12.1.1 API reference for networks.Network

class networks.**Network**(**layers: List[rockpool.layers.layer.Layer], dt: Optional[float] = None*)

Bases: object

Base class to manage networks (collections of *Layer* objects)

Network objects allow you to encapsulate a stack of layers with various configurations. Using a *Network* object allows you to connect layers in an acyclic graph, and verifies that adjacent layers have compatible sizes and signal classes.

Example of building a network

Specify the network sizes

```
>>> input_size = 2
>>> reservoir_size = 10
>>> output_size = 1
```

Generate layer weights

```
>>> weights_in = np.random.rand(input_size, reservoir_size)
>>> weights_rec = np.random.randn(reservoir_size, reservoir_size)
>>> weights_out = np.random.rand(reservoir_size, output_size)
```

Generate the layers

```
>>> lyr_in = FFRateEuler(weights_in)
>>> lyr_rec = RecRateEuler(weights_rec)
>>> lyr_out = PassThrough(weights_out)
```

Generate the *Network* object

```
>>> net = Network([lyr_in, lyr_rec, lyr_out])
```

See also:

The tutorial *Building and simulating a reservoir network* illustrates using a `Network` object to encapsulate a reservoir network.

__init__(**layers: List[rockpool.layers.layer.Layer], dt: Optional[float] = None*)
Base class to encapsulate several `Layer` objects and manage signal routing

Parameters

- **layers** (*Iterable[Layer]*) – Layers to be added to the network. They will be connected in series. The order in which they are received determines the order in which they are connected. First layer will receive external input
- **dt** (*Optional[float]*) – If not none, network time step is forced to this values. Layers that are added must have time step that is multiple of dt. If None, network will try to determine suitable dt each time a layer is added.

Attributes

dt	(float) Time step to use in layer simulations
t	(float) Global network time

Methods

__init__(*layers[, dt])	Base class to encapsulate several `Layer` objects and manage signal routing
add_layer(lyr[, input_layer, output_layer, ...])	Add a new layer to the network
add_layer_class(cls_lyr, name)	Add external layer class to the namespace
connect(pre_layer, post_layer[, verbose])	Connect two layers by defining one as the input layer of the other
disconnect(pre_layer, post_layer[, verbose])	Remove the connection between two layers by setting the input of the target layer to `None`
evolve([ts_input, duration, num_timesteps, ...])	Evolve the network by evolving each layer in turn
load(filename)	Load a network from a JSON file
remove_layer(del_layer)	Remove a layer from the network by removing it from the layer inventory and make sure that no other layer receives input from it
reset_all()	Reset all state and time of the network and layers
reset_state()	Reset the state of the network by resetting each layer.
reset_time()	Reset the time of the network to zero by resetting each layer and the global network timestamp.
save(filename)	Save this network to a JSON file
shallow_copy()	shallow_copy - Generate and return a `Network` of the same structure with
stream(ts_input[, duration, num_timesteps, ...])	Stream data through layers, evolving by single time steps
train(training_fct[, ts_input, duration, ...])	Train the network batch-wise by evolving the layers and calling the training function

__init__(**layers: List[rockpool.layers.layer.Layer], dt: Optional[float] = None*)
Base class to encapsulate several *Layer* objects and manage signal routing

Parameters

- **layers** (*Iterable[Layer]*) – Layers to be added to the network. They will be connected in series. The order in which they are received determines the order in which they are connected. First layer will receive external input
- **dt** (*Optional[float]*) – If not none, network time step is forced to this values. Layers that are added must have time step that is multiple of dt. If None, network will try to determine suitable dt each time a layer is added.

_check_sync(*verbose: bool = True*) → bool
Check whether the time *t* of all layers matches `self.t`. If not, raise an exception

Parameters **verbose** (*Optional[bool]*) – If `True`, display feedback. Default: `True`, display feedback.

Raises **NetworkError** – If layers are not in synch with global network time

_fix_duration(*t: float*) → float
Correct an evolution duration so that it is a multiple of *dt*

Due to rounding errors it can happen that a duration or end time *t* is slightly below its intened value, causing the layers to not evolve sufficiently. This method fixes the problem by increasing *t* if it is slightly below a multiple of *dt* of any of the layers in the network.

Parameters **t** (*float*) – Time to be fixed

Return float Corrected duration

static _new_name(*name: str*) → str
Generate a new name by first checking whether the old name ends with '_i', with i an integer. If so, replace i by i+1, otherwise append '_0'

Parameters **name** (*str*) – Name to be modified

Return str Modified name

_set_dt(*max_factor: float = 100*)
Set a time step size for the network which is the lcm of all layers' dt's.

Parameters **max_factor** (*float*) – Factor by which the network *dt* may exceed the largest layer *Layer.dt* before an error is raised

Raises **ValueError** – If a sensible *dt* cannot be found

_set_evolution_order() → list
Determine the order in which layers are evolved. Requires Network to be a directed acyclic graph, otherwise evolution has to happen timestep-wise instead of layer-wise

add_layer(*lyr: rockpool.layers.layer.Layer*, *input_layer: rockpool.layers.layer.Layer = None*, *output_layer: rockpool.layers.layer.Layer = None*, *external_input: bool = False*, *verbose: bool = False*) → rockpool.layers.layer.Layer
Add a new layer to the network

Add `lyr` to `self` and to `layerset`. Its attribute name is 'lyr'+lyr.name. Check whether layer with this name already exists (replace anyway). Connect `lyr` to `input_layer` and `output_layer`.

Parameters

- **lyr** (*Layer*) – layer to be added to the network
- **input_layer** (*Optional[Layer]*) – Layer to connect as an input layer to `lyr`

- **output_layer** (*Optional[Layer]*) – Layer to connect as an output layer from `lyr`
- **external_input** (*Optional[bool]*) – If `True`, this layer should receive external input. Default: `False`, `lyr` should not be connected to external input
- **verbose** (*Optional[bool]*) – If `True`, print feedback about layer addition. Default: `False`, do not display feedback

Return Layer `lyr`, the connected layer

static add_layer_class(*cls_lyr: Type[rockpool.layers.layer.Layer], name: str*)
Add external layer class to the namespace

This method adds a externally-defined *Layer* subclass to the `layers` namespace, so that layers that are defined outside the `rockpool.layers` module can still be loaded

Parameters

- **cls** (Layer) – The class that is to be added
- **name** (*str*) – Name of the class as a string

connect(*pre_layer: rockpool.layers.layer.Layer*, *post_layer: rockpool.layers.layer.Layer*, *verbose: bool = False*)
Connect two layers by defining one as the input layer of the other

Parameters

- **pre_layer** (Layer) – The source layer
- **post_layer** (Layer) – The target layer
- **verbose** (*Optional[bool]*) – If `True`, display feedback about the connection process. Default: `False`, do not display feedback

Raises **NetworkError** – if layers do not have compatible output / input sizes, or incompatible time series classes

disconnect(*pre_layer: rockpool.layers.layer.Layer*, *post_layer: rockpool.layers.layer.Layer*, *verbose: bool = False*)
Remove the connection between two layers by setting the input of the target layer to `None`

Parameters

- **pre_layer** (Layer) – The source layer
- **post_layer** (Layer) – The target layer
- **verbose** (*Optional[bool]*) – If `True`, display feedback about the connection process. Default: `False`, do not display feedback

property dt
(float) Time step to use in layer simulations

evolve(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = True*) → dict
Evolve the network by evolving each layer in turn

Evolve each layer in the network according to self.evol_order. For layers with external_input==True their input is ts_input. If not but an input layer is defined, it will be the output of that, otherwise None. Return a dict with each layer's output.

See also:

Getting started with Rockpool and the tutorial *Building and simulating a reservoir network* show examples of using the *evolve* method.

Parameters

- **ts_input** (*Optional[TimeSeries]*) – External input to the network. Default: None, no external input
- **duration** (*Optional[float]*) – Duration over which network should be evolved. If not provided, then num_timesteps or the duration of ts_input will determine the evolution duration
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then duration of the duration of ts_input will determine evolution duration
- **verbose** (*Optional[bool]*) – If True, display info about evolution state. Default: True, display feedback

Return dict Dictionary containing the output time series of each layer. Entries in the dictionary will be have keys taken from the names of each layer

Raises **AssertionError** – If no duration can be determined

static load(*filename: str*) → rockpool.networks.network.Network
Load a network from a JSON file

Parameters **filename** (*str*) – filename of a JSON filr that contains a saved network

Return Network A network object with all the layers loaded from filename

remove_layer(*del_layer: rockpool.layers.layer.Layer*)
Remove a layer from the network by removing it from the layer inventory and make sure that no other layer receives input from it

Parameters **del_layer** (Layer) – Layer to be removed from the network

reset_all()
Reset all state and time of the network and layers

reset_state()
Reset the state of the network by resetting each layer. Does not reset time.

reset_time()
Reset the time of the network to zero by resetting each layer and the global network timestamp. Does not reset state.

save(*filename: str*)
Save this network to a JSON file

Parameters **filename** (*str*) – The path to a file in which to save the network and state.

shallow_copy() → rockpool.networks.network.Network

shallow_copy - Generate and return a *Network* of the same structure with the *same* layer objects.

Returns The new *Network* object.

stream(*ts_input: rockpool.timeseries.TimeSeries*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*, *step_callback: Optional[Callable] = None*) → dict
Stream data through layers, evolving by single time steps

Parameters

- **ts_input** (TimeSeries) – External input to the network

- **duration** (*Optional[float]*) – Total duration to stream for. If not provided, use `num_timesteps` or the duration of `ts_input` to determine duration
- **num_timesteps** (*Optional[int]*) – Number of time steps to stream for, in units of *dt*. If not provided, using `duration` of the duration of `ts_input` to determine duration
- **verbose** (*Optional[bool]*) – If `True`, display feedback during streaming. Default: `False`, do not display feedback
- **step_callback** (*Optional[Callable]*) – Callback function that will be called on each time step. Has the signature Callable[[Network]]

Return dict Collected output signals from each layer

property t
(float) Global network time

train(*training_fct: Callable[[Network, Dict[str, rockpool.timeseries.TimeSeries], bool, bool], Any], ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, batch_durs: Union[numpy.ndarray, float, None] = None, num_timesteps: int = None, nums_ts_batch: Union[numpy.ndarray, int, None] = None, verbose: bool = True, high_verbosity: bool = False*)
Train the network batch-wise by evolving the layers and calling the training function

See also:

The tutorial *Building and simulating a reservoir network* illustrates how to call *train* and how to build a training function.

Parameters

- **training_fct** (*Callable*) – Function that is called after each evolution, taking the following arguments: - `net` (*Network*): Network the network object to be trained. - `signals` (`Dict`): Dictionary containing all signals in the current evolution batch. - `is_first` (`bool`): Is this the first batch? - `is_last` (`bool`): Is this the final batch?
- **ts_input** (*Optional[TimeSeries]*) – Time series containing external input to network
- **duration** (*Optional[float]*) – Duration over which network should be evolved. If None, evolution is over the duration of ts_input
- **batch_durs** (*Optional[ArrayLike[float]]*) – Array-like or float - Duration of one batch (can also pass array with several values)
- **num_timesteps** (*Optional[int]*) – Total number of training time steps
- **nums_ts_batch** (*Optional[ArrayLike[int]]*) – Array-like or int - Number of time steps per batch (or array of several values)
- **verbose** (*Optional[bool]*) – If `True`, print info about training progress. Default: `True`, display progress
- **high_verbosity** (*Optional[bool]*) – If `True`, print info about layer evolution (only has effect if `verbose` is `True`) Default: `False`, dont' display extra feedback

12.1.2 API reference for layers.Layer

class layers.**Layer**(*weights: numpy.ndarray*, *dt: Optional[float] = 1*, *noise_std: Optional[float] = 0*, *name: Optional[str] = 'unnamed'*)

Bases: abc.ABC

Base class for Layers in rockpool

This abstract class acts as a base class from which to derive subclasses that represent layers of neurons. As an abstract class, *Layer* cannot be instantiated.

See also:

See *Types of Layer available in Rockpool* for examples of instantiating and using *Layer* subclasses. See "Writing a new Layer subclass" for how to design and implement a new *Layer* subclass.

__init__(*weights: numpy.ndarray*, *dt: Optional[float] = 1*, *noise_std: Optional[float] = 0*, *name: Optional[str] = 'unnamed'*)

Implement an abstract layer of neurons (no implementation, must be subclasses)

Parameters

- **weights** (*ArrayLike[float]*) – Weight matrix for this layer. Indexed as [pre, post]
- **dt** (*float*) – Time-step used for evolving this layer. Default: 1
- **noise_std** (*float*) – Std. Dev. of state noise when evolving this layer. Default: 0. Defined as the expected std. dev. after 1s of integration time
- **name** – str Name of this layer. Default: 'unnamed'

Attributes

class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer name attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, dt, noise_std, name])	Implement an abstract layer of neurons (no implementation, must be subclasses)
evolve([ts_input, duration, num_timesteps])	Abstract method to evolve the state of this layer
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__ (*weights: numpy.ndarray*, *dt: Optional[float] = 1*, *noise_std: Optional[float] = 0*, *name: Optional[str] = 'unnamed'*)
Implement an abstract layer of neurons (no implementation, must be subclasses)

Parameters

- **weights** (*ArrayLike[float]*) – Weight matrix for this layer. Indexed as [pre, post]
- **dt** (*float*) – Time-step used for evolving this layer. Default: 1
- **noise_std** (*float*) – Std. Dev. of state noise when evolving this layer. Default: 0. Defined as the expected std. dev. after 1s of integration time
- **name** – str Name of this layer. Default: 'unnamed'

_check_input_dims (*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

abstract evolve(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → rockpool.timeseries.TimeSeries
Abstract method to evolve the state of this layer

This method must be overridden to produce a concrete *Layer* subclass. The *evolve* method is the main interface for simulating a layer. It must accept an input time series which determines the signals injected into the layer as input, and return an output time series representing the output of the layer.

Parameters

- **ts_input** (*Optional[TimeSeries]*) – (TxM) External input trace to use when evolving the layer
- **duration** (*Optional[float]*) – Duration in seconds to evolve the layer. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of time steps to evolve the layer, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time

Return TimeSeries (TxN) Output of this layer

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

abstract to_dict() → dict
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.1.3 Layer and Network alternative base classes

networks.NetworkDeneve()	*Network* base class that defines and builds networks of Deneve reservoirs to solve linear problems.
layers.training.RRTrainedLayer(weights[, ...])	Base class that defines methods for training with ridge regression.

API reference for networks.NetworkDeneve

class networks.**NetworkDeneve**
Bases: `rockpool.networks.network.Network`

Network base class that defines and builds networks of Deneve reservoirs to solve linear problems.

__init__()
Base class to encapsulate several *Layer* objects and manage signal routing

Parameters

- **layers** (*Iterable[Layer]*) – Layers to be added to the network. They will be connected in series. The order in which they are received determines the order in which they are connected. First layer will receive external input
- **dt** (*Optional[float]*) – If not none, network time step is forced to this values. Layers that are added must have time step that is multiple of dt. If None, network will try to determine suitable dt each time a layer is added.

Attributes

dt	(float) Time step to use in layer simulations
t	(float) Global network time

Methods

SolveLinearProblem([a, net_size, gamma, dt, ...])	SolveLinearProblem - Static method Direct implementation of a linear dynamical system

Continued on next page

Table 8 – continued from previous page

SpecifyNetwork(weights_fast, weights_slow, ...)	SpecifyNetwork - Directly configure all layers of a reservoir
__init__()	Base class to encapsulate several *Layer* objects and manage signal routing
add_layer(lyr[, input_layer, output_layer, ...])	Add a new layer to the network
add_layer_class(cls_lyr, name)	Add external layer class to the namespace
connect(pre_layer, post_layer[, verbose])	Connect two layers by defining one as the input layer of the other
disconnect(pre_layer, post_layer[, verbose])	Remove the connection between two layers by setting the input of the target layer to `None`
evolve([ts_input, duration, num_timesteps, ...])	Evolve the network by evolving each layer in turn
load(filename)	Load a network from a JSON file
remove_layer(del_layer)	Remove a layer from the network by removing it from the layer inventory and make sure that no other layer receives input from it
reset_all()	Reset all state and time of the network and layers
reset_state()	Reset the state of the network by resetting each layer.
reset_time()	Reset the time of the network to zero by resetting each layer and the global network timestamp.
save(filename)	Save this network to a JSON file
shallow_copy()	shallow_copy - Generate and return a *Network* of the same structure with
stream(ts_input[, duration, num_timesteps, ...])	Stream data through layers, evolving by single time steps
train(training_fct[, ts_input, duration, ...])	Train the network batch-wise by evolving the layers and calling the training function

static SolveLinearProblem(*a: numpy.ndarray = None*, *net_size: int = None*, *gamma: numpy.ndarray = None*, *dt: float = None*, *mu: float = 0.0001*, *nu: float = 0.001*, *noise_std: float = 0.0*, *tau_mem: float = 0.02*, *tau_syn_fast: float = 0.001*, *tau_syn_slow: float = 0.1*, *v_thresh: numpy.ndarray = -0.055*, *v_rest: numpy.ndarray = -0.065*, *refractory=-2.220446049250313e-16*)

SolveLinearProblem - Static method Direct implementation of a linear dynamical system

Parameters

- **a** – np.ndarray [PxP] Matrix defining linear dynamical system
- **net_size** – int Desired size of recurrent reservoir layer (Default: 100)
- **gamma** – np.ndarray [PxN] Input kernel (Default: 50 * Normal / net_size)
- **dt** – float Nominal time step (Default: 0.1 ms)
- **mu** – float Linear cost parameter (Default: 1e-4)
- **nu** – float Quadratic cost parameter (Default: 1e-3)
- **noise_std** – float Noise std. dev. (Default: 0)
- **tau_mem** – float Neuron membrane time constant (Default: 20 ms)
- **tau_syn_fast** – float Fast synaptic time constant (Default: 1 ms)
- **tau_syn_slow** – float Slow synaptic time constant (Default: 100 ms)
- **v_thresh** – float Threshold membrane potential (Default: -55 mV)

- **v_rest** – float Rest potential (Default: -65 mV)
- **refractory** – float Refractory time for neuron spiking (Default: 0 ms)

Returns A configured NetworkDeneve object, containing input, reservoir and output layers

static SpecifyNetwork(*weights_fast*, *weights_slow*, *weights_in*, *weights_out*, *dt: float = None*, *noise_std: float = 0.0*, *v_thresh: numpy.ndarray = -0.055*, *v_reset: numpy.ndarray = -0.065*, *v_rest: numpy.ndarray = -0.065*, *tau_mem: float = 0.02*, *tau_syn_r_fast: float = 0.001*, *tau_syn_r_slow: float = 0.1*, *tau_syn_out: float = 0.1*, *refractory: float = -2.220446049250313e-16*)

SpecifyNetwork - Directly configure all layers of a reservoir

Parameters

- **weights_fast** – np.ndarray [NxN] Matrix of fast synaptic weights
- **weights_slow** – np.ndarray [NxN] Matrix of slow synaptic weights
- **weights_in** – np.ndarray [LxN] Matrix of input kernels
- **weights_out** – np.ndarray [NxM] Matrix of output kernels
- **dt** – float Nominal time step
- **noise_std** – float Noise Std. Dev.
- **v_rest** – np.ndarray [Nx1] Vector of rest potentials (spiking layer)
- **v_reset** – np.ndarray [Nx1] Vector of reset potentials (spiking layer)
- **v_thresh** – np.ndarray [Nx1] Vector of threshold potentials (spiking layer)
- **tau_mem** – float Neuron membrane time constant (spiking layer)
- **tau_syn_r_fast** – float Fast recurrent synaptic time constant
- **tau_syn_r_slow** – float Slow recurrent synaptic time constant
- **tau_syn_out** – float Synaptic time constant for output layer
- **refractory** – float Refractory time for spiking layer

Returns

__init__()

Base class to encapsulate several *Layer* objects and manage signal routing

Parameters

- **layers** (*Iterable[Layer]*) – Layers to be added to the network. They will be connected in series. The order in which they are received determines the order in which they are connected. First layer will receive external input
- **dt** (*Optional[float]*) – If not none, network time step is forced to this values. Layers that are added must have time step that is multiple of dt. If None, network will try to determine suitable dt each time a layer is added.

_check_sync(*verbose: bool = True*) → bool

Check whether the time *t* of all layers matches `self.t`. If not, raise an exception

Parameters **verbose** (*Optional[bool]*) – If `True`, display feedback. Default: `True`, display feedback.

Raises **NetworkError** – If layers are not in synch with global network time

_fix_duration(*t: float*) → float
Correct an evolution duration so that it is a multiple of *dt*

Due to rounding errors it can happen that a duration or end time *t* is slightly below its intened value, causing the layers to not evolve sufficiently. This method fixes the problem by increasing *t* if it is slightly below a multiple of *dt* of any of the layers in the network.

Parameters **t** (*float*) – Time to be fixed

Return float Corrected duration

static _new_name(*name: str*) → str
Generate a new name by first checking whether the old name ends with '_i', with i an integer. If so, replace i by i+1, otherwise append '_0'

Parameters **name** (*str*) – Name to be modified

Return str Modified name

_set_dt(*max_factor: float = 100*)
Set a time step size for the network which is the lcm of all layers' dt's.

Parameters **max_factor** (*float*) – Factor by which the network *dt* may exceed the largest layer *Layer.dt* before an error is raised

Raises **ValueError** – If a sensible *dt* cannot be found

_set_evolution_order() → list
Determine the order in which layers are evolved. Requires Network to be a directed acyclic graph, otherwise evolution has to happen timestep-wise instead of layer-wise

add_layer(*lyr: rockpool.layers.layer.Layer*, *input_layer: rockpool.layers.layer.Layer = None*, *output_layer: rockpool.layers.layer.Layer = None*, *external_input: bool = False*, *verbose: bool = False*) → rockpool.layers.layer.Layer
Add a new layer to the network

Add `lyr` to `self` and to `layerset`. Its attribute name is 'lyr'+lyr.name. Check whether layer with this name already exists (replace anyway). Connect `lyr` to `input_layer` and `output_layer`.

Parameters

- **lyr** (*Layer*) – layer to be added to the network
- **input_layer** (*Optional[Layer]*) – Layer to connect as an input layer to `lyr`
- **output_layer** (*Optional[Layer]*) – Layer to connect as an output layer from `lyr`
- **external_input** (*Optional[bool]*) – If `True`, this layer should receive external input. Default: `False`, `lyr` should not be connected to external input
- **verbose** (*Optional[bool]*) – If `True`, print feedback about layer addition. Default: `False`, do not display feedback

Return Layer `lyr`, the connected layer

static add_layer_class(*cls_lyr: Type[rockpool.layers.layer.Layer], name: str*)
Add external layer class to the namespace

This method adds a externally-defined *Layer* subclass to the `layers` namespace, so that layers that are defined outside the `rockpool.layers` module can still be loaded

Parameters

- **cls** (*Layer*) – The class that is to be added
- **name** (*str*) – Name of the class as a string

connect (*pre_layer: rockpool.layers.layer.Layer*, *post_layer: rockpool.layers.layer.Layer*, *verbose: bool = False*)
Connect two layers by defining one as the input layer of the other

Parameters

- **pre_layer** (Layer) – The source layer
- **post_layer** (Layer) – The target layer
- **verbose** (*Optional[bool]*) – If True, display feedback about the connection process. Default: False, do not display feedback

Raises **NetworkError** – if layers do not have compatible output / input sizes, or incompatible time series classes

disconnect (*pre_layer: rockpool.layers.layer.Layer*, *post_layer: rockpool.layers.layer.Layer*, *verbose: bool = False*)
Remove the connection between two layers by setting the input of the target layer to None

Parameters

- **pre_layer** (Layer) – The source layer
- **post_layer** (Layer) – The target layer
- **verbose** (*Optional[bool]*) – If True, display feedback about the connection process. Default: False, do not display feedback

property dt
(float) Time step to use in layer simulations

evolve (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = True*) → dict
Evolve the network by evolving each layer in turn

Evolve each layer in the network according to self.evol_order. For layers with external_input==True their input is ts_input. If not but an input layer is defined, it will be the output of that, otherwise None. Return a dict with each layer's output.

See also:

Getting started with Rockpool and the tutorial *Building and simulating a reservoir network* show examples of using the *evolve* method.

Parameters

- **ts_input** (*Optional[TimeSeries]*) – External input to the network. Default: None, no external input
- **duration** (*Optional[float]*) – Duration over which network should be evolved. If not provided, then num_timesteps or the duration of ts_input will determine the evolution duration
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then duration of the duration of ts_input will determine evolution duration
- **verbose** (*Optional[bool]*) – If True, display info about evolution state. Default: True, display feedback

Return dict Dictionary containing the output time series of each layer. Entries in the dictionary will be have keys taken from the names of each layer

Raises **AssertionError** – If no duration can be determined

static load(*filename: str*) → rockpool.networks.network.Network
Load a network from a JSON file

Parameters **filename** (*str*) – filename of a JSON filr that contains a saved network

Return Network A network object with all the layers loaded from `filename`

remove_layer(*del_layer: rockpool.layers.layer.Layer*)
Remove a layer from the network by removing it from the layer inventory and make sure that no other layer receives input from it

Parameters **del_layer** (`Layer`) – Layer to be removed from the network

reset_all()
Reset all state and time of the network and layers

reset_state()
Reset the state of the network by resetting each layer. Does not reset time.

reset_time()
Reset the time of the network to zero by resetting each layer and the global network timestamp. Does not reset state.

save(*filename: str*)
Save this network to a JSON file

Parameters **filename** (*str*) – The path to a file in which to save the network and state.

shallow_copy() → rockpool.networks.network.Network

shallow_copy - Generate and return a *Network* of the same structure with the *same* layer objects.

Returns The new *Network* object.

stream(*ts_input: rockpool.timeseries.TimeSeries*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*, *step_callback: Optional[Callable] = None*) → dict
Stream data through layers, evolving by single time steps

Parameters

- **ts_input** (`TimeSeries`) – External input to the network
- **duration** (*Optional[float]*) – Total duration to stream for. If not provided, use `num_timesteps` or the duration of `ts_input` to determine duration
- **num_timesteps** (*Optional[int]*) – Number of time steps to stream for, in units of *dt*. If not provided, using `duration` of the duration of `ts_input` to determine duration
- **verbose** (*Optional[bool]*) – If `True`, display feedback during streaming. Default: `False`, do not display feedback
- **step_callback** (*Optional[Callable]*) – Callback function that will be called on each time step. Has the signature Callable[[Network]]

Return dict Collected output signals from each layer

property t
(float) Global network time

train(*training_fct: Callable[[Network, Dict[str, rockpool.timeseries.TimeSeries], bool, bool], Any], ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, batch_durs: Union[numpy.ndarray, float, None] = None, num_timesteps: int = None, nums_ts_batch: Union[numpy.ndarray, int, None] = None, verbose: bool = True, high_verbosity: bool = False*)

Train the network batch-wise by evolving the layers and calling the training function

See also:

The tutorial *Building and simulating a reservoir network* illustrates how to call `train` and how to build a training function.

Parameters

- **training_fct** (*Callable*) – Function that is called after each evolution, taking the following arguments: - `net` (*Network*): Network the network object to be trained. - `signals` (`Dict`): Dictionary containing all signals in the current evolution batch. - `is_first` (`bool`): Is this the first batch? - `is_last` (`bool`): Is this the final batch?
- **ts_input** (*Optional[TimeSeries]*) – Time series containing external input to network
- **duration** (*Optional[float]*) – Duration over which network should be evolved. If None, evolution is over the duration of ts_input
- **batch_durs** (*Optional[ArrayLike[float]]*) – Array-like or float - Duration of one batch (can also pass array with several values)
- **num_timesteps** (*Optional[int]*) – Total number of training time steps
- **nums_ts_batch** (*Optional[ArrayLike[int]]*) – Array-like or int - Number of time steps per batch (or array of several values)
- **verbose** (*Optional[bool]*) – If `True`, print info about training progress. Default: `True`, display progress
- **high_verbosity** (*Optional[bool]*) – If `True`, print info about layer evolution (only has effect if `verbose` is `True`) Default: `False`, dont' display extra feedback

API reference for layers.training.RRTrainedLayer

class layers.training.**RRTrainedLayer**(*weights: numpy.ndarray, dt: Optional[float] = 1, noise_std: Optional[float] = 0, name: Optional[str] = 'unnamed'*)

Bases: `rockpool.layers.layer.Layer`, `abc.ABC`

Base class that defines methods for training with ridge regression. Subclasses can inherit from this class to provide ridge regression functionality.

Usage

When writing a new layer class, simply inherit from *RRTrainedLayer* instead of from *Layer*. Subclasses must provide a concrete implementation of the the *_prepare_training_data* abstract method. See the documentation for that method below, to understand how this can be implemented. *RRTrainedLayer* provides an implementation that can be called with `super()`.

This class provides the *train_rr* method, which performs ridge regression training over multiple batches, called independently for each batch.

RRTrainedLayer also provides the *_batch_update* private method

__init__(*weights: numpy.ndarray*, *dt: Optional[float] = 1*, *noise_std: Optional[float] = 0*, *name: Optional[str] = 'unnamed'*)
Implement an abstract layer of neurons (no implementation, must be subclasses)

Parameters

- **weights** (*ArrayLike[float]*) – Weight matrix for this layer. Indexed as [pre, post]
- **dt** (*float*) – Time-step used for evolving this layer. Default: 1
- **noise_std** (*float*) – Std. Dev. of state noise when evolving this layer. Default: 0. Defined as the expected std. dev. after 1s of integration time
- **name** – str Name of this layer. Default: 'unnamed'

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, dt, noise_std, name])	Implement an abstract layer of neurons (no implementation, must be subclasses)
evolve([ts_input, duration, num_timesteps])	Abstract method to evolve the state of this layer
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file

Continued on next page

Table 10 – continued from previous page

save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer
train_rr(ts_target[, ts_input, regularize, ...])	Train this layer with ridge regression over one of possibly many batches.

__init__(*weights: numpy.ndarray*, *dt: Optional[float] = 1*, *noise_std: Optional[float] = 0*, *name: Optional[str] = 'unnamed'*)
Implement an abstract layer of neurons (no implementation, must be subclasses)

Parameters

- **weights** (*ArrayLike[float]*) – Weight matrix for this layer. Indexed as [pre, post]
- **dt** (*float*) – Time-step used for evolving this layer. Default: 1
- **noise_std** (*float*) – Std. Dev. of state noise when evolving this layer. Default: 0. Defined as the expected std. dev. after 1s of integration time
- **name** – str Name of this layer. Default: 'unnamed'

_batch_update(*inp: numpy.ndarray*, *target: numpy.ndarray*, *reset: bool*, *train_biases: bool*, *standardize: bool*, *update_weights: bool*, *return_training_progress: bool*) → Dict
Train with the already processed input and target data of the current batch. Update layer weights and biases if requested. Provide information on training state if requested.

Parameters

- **inp** (*np.ndarray*) – 2D-array (num_samples x num_features) of input data.
- **target** (*np.ndarray*) – 2D-array (num_samples x 'self.size') of target data.
- **reset** (*bool*) – If 'True', internal variables will be reset at the end.
- **train_bises** (*bool*) – Should biases be trained or only weights?
- **standardize** (*bool*) – Has input data been z-score standardized?
- **update_weights** (*bool*) – Set 'True' to update layer weights and biases.
- **return_training_progress** (*bool*) – Return intermediate training data (e.g. xtx, xty,...)

Return dict Dict with information on training progress, depending on values of other function arguments.

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

abstract _prepare_training_data (*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TimeSeries*, *is_first: bool*, *is_last: bool*) -> (typing.Union[NoneType, numpy.ndarray], <class 'numpy.ndarray'>, <class 'numpy.ndarray'>)

Template for preparation of training data. Length of data is determined, dimensions are verified and target data is extracted from `ts_target` argument. Can be used in child classes through `super()._prepare_training_data`. Extraction of input data needs to be implemented in child classes.

Parameters

- **ts_target** (`TSContinuous`) – Target time series
- **ts_input** (`TimeSeries`) – Input time series
- **is_first** (*bool*) – Set `True` if batch is first of training.
- **is_last** (*bool*) – Set `True` if batch is last of training.

Return (input, target, time_base) input np.ndarray: Should be extracted input data. This abstract method returns `None` target np.ndarray: Extracted target data time_base np.ndarray: Time base for the input and target data

Usage

Child classes must implement this method to be instantiated. However, the abstract method provided here performs several checks and useful functions:

```
# - In the child class, call the superclass method
__, target, time_base = super()._prepare_training_data(ts_target, ts_input,␣
↪is_first, is_last)

# ... perform input extraction from ``ts_input`` here in child class
```

property class_name

(str) Class name of `self`

property dt
(float) Simulation time step of this layer

abstract evolve(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → rockpool.timeseries.TimeSeries
Abstract method to evolve the state of this layer

This method must be overridden to produce a concrete `Layer` subclass. The *evolve* method is the main interface for simulating a layer. It must accept an input time series which determines the signals injected into the layer as input, and return an output time series representing the output of the layer.

Parameters

- **ts_input** (*Optional[TimeSeries]*) – (TxM) External input trace to use when evolving the layer
- **duration** (*Optional[float]*) – Duration in seconds to evolve the layer. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of time steps to evolve the layer, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time

Return TimeSeries (TxN) Output of this layer

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing `Layer`, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

abstract to_dict() → dict
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

train_rr(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Union[rockpool.timeseries.TSEvent, rockpool.timeseries.TSContinuous, None] = None*, *regularize: Optional[float] = 0*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*, *train_biases: Optional[bool] = True*, *calc_intermediate_results: Optional[bool] = False*, *return_training_progress: Optional[bool] = True*, *return_trained_output: Optional[bool] = False*, *fisher_relabelling: Optional[bool] = False*, *standardize: Optional[bool] = False*) → Optional[Dict]
Train this layer with ridge regression over one of possibly many batches. Use Kahan summation to reduce rounding errors when adding data to existing matrices from previous batches.

Parameters

- **ts_target** (*TSContinuous*) – Target signal for current batch
- **ts_input** (*Optional[TimeSeries]*) – Input to layer for current batch. Default: `None`, no input for this batch
- **regularize** (*Optional[float]*) – Regularization parameter for ridge regression. Default: `0`, no regularization
- **is_first** (*Optional[bool]*) – Set to `True` if current batch is the first in training. Default: `True`, initialise training with this batch as the first batch
- **is_last** (*Optional[bool]*) – Set to `True` if current batch is the last in training. This has the same effect as if data from both trainings were presented at once.
- **train_biases** (*Optional[bool]*) – If `True`, train biases as if they were weights. Otherwise present biases will be ignored in training and not be changed. Default: `True`, train biases as well as weights
- **calc_intermediate_results** (*Optional[bool]*) – If `True`, calculates the intermediate weights not in the final batch. Default: `False`, do not compute intermediate weights
- **return_training_progress** (*Optional[bool]*) – If `True`, return dict of current training variables for each batch. Default: `True`, return training progress
- **return_trained_output** (*Optional[bool]*) – If `True`, return the result of evolving the layer with the trained weights in the output dict. Default: `False`, do not return the trained output
- **fisher_relabelling** (*Optional[bool]*) – If `True`, relabel target data such that the training algorithm is equivalent to Fisher discriminant analysis. Default: `False`, use standard ridge / linear regression
- **standardize** (*Optional[bool]*) – Train with z-score standardized data, based on means and standard deviations from first batch. Default: `False`, do not standardize data

Returns If `return_training_progress` is `True`, return a dict with current training variables (xtx, xty, kahan_comp_xtx, kahan_comp_xty). Weights and biases are returned if `is_last` is `True` or if `calc_intermediate_results` is `True`. If `return_trained_output` is `True`, the dict contains the output of evolving the layer with the newly trained weights.

property weights

(ndarray) Weights encapsulated by this layer (MxN)

12.2 Time series classes

See also:

Working with time series data.

timeseries.TimeSeries(times[, periodic, ...])	Super-class to represent a continuous or event-based time series.
timeseries.TSContinuous([times, samples, ...])	Represents a continuously-sampled time series.

Continued on next page

Table 11 – continued from previous page

timeseries.TSEvent([times, channels, ...])	Represents a discrete time series, composed of binary events (present or absent).

12.2.1 API reference for timeseries.TimeSeries

class timeseries.**TimeSeries**(*times: Union[numpy.ndarray, List, Tuple], periodic: bool = False, t_start: Optional[float] = None, t_stop: Optional[float] = None, plotting_backend: Optional[str] = None, name: str = 'unnamed'*)

Bases: object

Super-class to represent a continuous or event-based time series. You should use the subclasses *TSContinuous* and *TSEvent* to represent continuous-time and event-based time series, respectively. See *Working with time series data* for futher explanation and examples.

__init__(*times: Union[numpy.ndarray, List, Tuple], periodic: bool = False, t_start: Optional[float] = None, t_stop: Optional[float] = None, plotting_backend: Optional[str] = None, name: str = 'unnamed'*)

TimeSeries - Represent a continuous or event-based time series

Parameters

- **times** (*ArrayLike*) – [Tx1] vector of time samples
- **periodic** (*bool*) – Treat the time series as periodic around the end points. Default: False
- **t_start** (*Optional[float]*) – If not None, the series start time is t_start, otherwise times[0]
- **t_stop** (*Optional[float]*) – If not None, the series stop time is t_stop, otherwise times[-1]
- **plotting_backend** (*Optional[str]*) – Determines plotting backend. If None, backend will be chosen automatically based on what is available.
- **name** (*str*) – Name of the TimeSeries object. Default: "unnamed"

Attributes

duration	(float) Duration of TimeSeries
plotting_backend	(str) Current plotting backend
t_start	(float) Start time of time series
t_stop	(float) Stop time of time series (final sample)
times	(ArrayLike[float]) Array of sample times

Methods

__init__(times[, periodic, t_start, t_stop, ...])	TimeSeries - Represent a continuous or event-based time series
copy()	Return a deep copy of this time series
delay(offset[, inplace])	Return a copy of self that is delayed by an offset
isempty()	Test if this TimeSeries object is empty
print()	print() - Print an overview of the time series.

Continued on next page

Table 13 – continued from previous page

set_plotting_backend(backend[, verbose])	Set which plotting backend to use with the .plot() method

__init__ (*times: Union[numpy.ndarray, List, Tuple], periodic: bool = False, t_start: Optional[float] = None, t_stop: Optional[float] = None, plotting_backend: Optional[str] = None, name: str = 'unnamed'*)
TimeSeries - Represent a continuous or event-based time series

Parameters

- **times** (*ArrayLike*) – [Tx1] vector of time samples
- **periodic** (*bool*) – Treat the time series as periodic around the end points. Default: `False`
- **t_start** (*Optional[float]*) – If not `None`, the series start time is `t_start`, otherwise `times[0]`
- **t_stop** (*Optional[float]*) – If not `None`, the series stop time is `t_stop`, otherwise `times[-1]`
- **plotting_backend** (*Optional[str]*) – Determines plotting backend. If `None`, backend will be chosen automatically based on what is available.
- **name** (*str*) – Name of the TimeSeries object. Default: "unnamed"

_modulo_period (*times: Union[numpy.ndarray, List, Tuple, float, int]*) → Union[numpy.ndarray, List, Tuple, float, int]
_modulo_period - Calculate provided times modulo `self.duration`

copy () → timeseries.TimeSeries
Return a deep copy of this time series

Return TimeSeries copy of `self`

delay (*offset: Union[int, float], inplace: bool = False*) → timeseries.TimeSeries
Return a copy of `self` that is delayed by an offset

For delaying self, use the `inplace` argument, or `.times += ...` instead.

Parameters

- **Offset** (*float*) – Time by which to offset this time series
- **inplace** (*bool*) – If `True`, conduct operation in-place (Default: `False`; create a copy)

Return TimeSeries New TimeSeries, delayed

property duration
(float) Duration of TimeSeries

isempty () → bool
Test if this TimeSeries object is empty

Return bool `True` iff the TimeSeries object contains no samples

property plotting_backend
(str) Current plotting backend

print ()
print() - Print an overview of the time series.

set_plotting_backend (*backend: Optional[str], verbose: bool = True*)
Set which plotting backend to use with the .plot() method

Parameters

- **backend** (*str*) – Specify a backend to use. Supported: {"holoviews", "matplotlib"}
- **verbose** (*bool*) – If True, print feedback about which backend has been set

property t_start
(float) Start time of time series

property t_stop
(float) Stop time of time series (final sample)

property times
(ArrayLike[float]) Array of sample times

12.2.2 API reference for timeseries.TSContinuous

class timeseries.**TSContinuous**(*times: Union[numpy.ndarray, List, Tuple, None] = None, samples: Union[numpy.ndarray, List, Tuple, None] = None, num_channels: Optional[int] = None, periodic: Optional[bool] = False, t_start: Optional[float] = None, t_stop: Optional[float] = None, name: Optional[str] = 'unnamed', units: Optional[str] = None, interp_kind: str = 'linear'*)

Bases: *timeseries.TimeSeries*

Represents a continuously-sampled time series. Mutliple time series can be represented by a single *TSContinuous* object, and have identical time bases. Temporal periodicity is supported. See *Working with time series data* for further explanation and examples.

Examples

Build a linearly-increasing time series that extends from 0 to 1 second

```
>>> time_base = numpy.linspace(0, 1, 100)
>>> samples = time_base
>>> ts = TSContinuous(time_base, samples)
```

Build a periodic time series as a sinusoid

```
>>> time_base = numpy.linspace(0, 2 * numpy.pi, 100)
>>> samples = numpy.sin(time_base)
>>> ts = TSContinuous(time_base, samples, periodic = True)
```

Build an object containing five random time series

```
>>> time_base = numpy.linspace(0, 1, 100)
>>> samples = numpy.random.rand((100, 5))
>>> ts = TSContinuous(time_base, samples)
```

Manipulate time series using standard operators

```
>>> ts + 5
>>> ts - 3
>>> ts * 2
>>> ts / 7
>>> ts // 3
>>> ts ** 2
>>> ts1 + ts2
...
```

Manipulate time series data in time

```
>>> ts.delay(4)
>>> ts.clip(start, stop, [channel1, channel2, channel3])
```

Combine time series data

```
>>> ts1.append_t(ts2)     # Appends the second time series, along the time axis
>>> ts1.append_c(ts2)     # Appends the second time series as an extra channel
```

Note: All *TSContinuous* manipulation methods return a copy by default. Most methods accept an optional `inplace` flag, which if `True` causes the operation to be performed in place.

Resample a time series using functional notation, list notation, or using the `resample()` method.

```
>>> ts(0.5)
>>> ts([0, .1, .2, .3])
>>> ts(numpy.array([0, .1, .2, .3]))
>>> ts[0.5]
>>> ts[0, .1, .2, .3]
>>> ts.resample(0.5)
>>> ts.resample([0, .1, .2, .3])
```

Resample using slice notation

```
>>> ts[0:.1:1]
```

Resample and select channels simultaneously

```
>>> ts[0:.1:1, :3]
```

__init__(*times: Union[numpy.ndarray, List, Tuple, None] = None*, *samples: Union[numpy.ndarray, List, Tuple, None] = None*, *num_channels: Optional[int] = None*, *periodic: Optional[bool] = False*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *name: Optional[str] = 'unnamed'*, *units: Optional[str] = None*, *interp_kind: str = 'linear'*)

TSContinuous - Represents a continuously-sample time series, supporting interpolation and periodicity.

Parameters

- **times** (*ArrayLike*) – [Tx1] vector of time samples
- **samples** (*ArrayLike*) – [TxM] matrix of values corresponding to each time sample
- **num_channels** (*Optional[in]*) – If `samples` is None, determines the number of channels of `self`. Otherwise it has no effect at all.
- **periodic** (*Optional[bool]*) – Treat the time series as periodic around the end points. Default: `False`
- **t_start** (*Optional[float]*) – If not `None`, the series start time is `t_start`, otherwise `times[0]`
- **t_stop** (*Optional[float]*) – If not `None`, the series stop time is `t_stop`, otherwise `times[-1]`
- **name** (*Optional[str]*) – Name of the *TSContinuous* object. Default: `"unnamed"`
- **units** (*Optional[str]*) – Units of the *TSContinuous* object. Default: `None`

- **interp_kind** (*Optional[str]*) – Specify the interpolation type. Default: "linear"

If the time series is not periodic (the default), then NaNs will be returned for any extrapolated values.

Attributes

duration	(float) Duration of TimeSeries
max	(float) Maximum value of time series
min	(float) Minimum value of time series
num_channels	(int) Number of channels (dimension of sample vectors) in this TimeSeries object
num_traces	(int) Synonymous to num_channels
plotting_backend	(str) Current plotting backend
samples	(ArrayLike[float]) Value of time series at sampled times
t_start	(float) Start time of time series
t_stop	(float) Stop time of time series (final sample)
times	(ArrayLike[float]) Array of sample times

Methods

__init__([times, samples, num_channels, ...])	TSContinuous - Represents a continuously-sample time series, supporting interpolation and periodicity.
append_c(other_series[, inplace])	Append another time series to this one, along the samples axis (i.e.
append_t(other_series[, offset, inplace])	Append another time series to this one, along the time axis
clip([t_start, t_stop, channels, ...])	Return a TSContinuous which is restricted to given time limits and only contains events of selected channels
contains(times)	Does the time series contain the time range specified in the given time trace? Always true for periodic series
copy()	Return a deep copy of this time series
delay(offset[, inplace])	Return a copy of self that is delayed by an offset
isempty()	Test if this TimeSeries object is empty
merge(other_series[, remove_duplicates, inplace])	Merge another time series to this one, by interleaving in time.
plot([times, target, channels, stagger, skip])	Visualise a time series on a line plot
print([full, num_first, num_last, limit_shorten])	Print an overview of the time series and its values
resample(times[, channels, inplace])	Return a new time series sampled to the supplied time base
save(path[, verbose])	Save this time series as an npz file using np.savez
set_plotting_backend(backend[, verbose])	Set which plotting backend to use with the .plot() method

__init__(*times: Union[numpy.ndarray, List, Tuple, None] = None*, *samples: Union[numpy.ndarray, List, Tuple, None] = None*, *num_channels: Optional[int] = None*, *periodic: Optional[bool] = False*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *name: Optional[str] = 'unnamed'*, *units: Optional[str] = None*, *interp_kind: str = 'linear'*)

TSContinuous - Represents a continuously-sample time series, supporting interpolation and periodicity.

Parameters

- **times** (*ArrayLike*) – [Tx1] vector of time samples
- **samples** (*ArrayLike*) – [TxM] matrix of values corresponding to each time sample
- **num_channels** (*Optional[int]*) – If `samples` is None, determines the number of channels of `self`. Otherwise it has no effect at all.
- **periodic** (*Optional[bool]*) – Treat the time series as periodic around the end points. Default: `False`
- **t_start** (*Optional[float]*) – If not `None`, the series start time is `t_start`, otherwise `times[0]`
- **t_stop** (*Optional[float]*) – If not `None`, the series stop time is `t_stop`, otherwise `times[-1]`
- **name** (*Optional[str]*) – Name of the *TSContinuous* object. Default: `"unnamed"`
- **units** (*Optional[str]*) – Units of the *TSContinuous* object. Default: `None`
- **interp_kind** (*Optional[str]*) – Specify the interpolation type. Default: `"linear"`

If the time series is not periodic (the default), then NaNs will be returned for any extrapolated values.

_compatible_shape(*other_samples*) → numpy.ndarray

Attempt to make `other_samples` a compatible shape to `self.samples`.

Parameters **other_samples** (*ArrayLike*) – Samples to convert

Return np.ndarray Array the same shape as `self.samples`

Raises ValueError if broadcast fails

_create_interpolator()

Build an interpolator for the samples in this TimeSeries.

Replaces the current interpolator.

_interpolate(*times: Union[int, float, numpy.ndarray, List, Tuple]*) → numpy.ndarray

Interpolate the time series to the provided time points

Parameters **times** (*ArrayLike*) – Array of `T` desired interpolated time points

Return np.ndarray Array of interpolated values. Will have the shape `TxN`, where `N` is the number of channels in `self`

_modulo_period(*times: Union[numpy.ndarray, List, Tuple, float, int]*) → Union[numpy.ndarray, List, Tuple, float, int]

_modulo_period - Calculate provided times modulo `self.duration`

append_c(*other_series: timeseries.TSContinuous*, *inplace: bool = False*) → timeseries.TSContinuous

Append another time series to this one, along the samples axis (i.e. add new channels)

Parameters

- **other_series** (TSContinuous) – Another time series. Will be resampled to the time base of `self`
- **inplace** (*bool*) – Conduct operation in-place (Default: `False`; create a copy)

Return TSContinuous*TSContinuous* Current time series, with new channels appended

append_t(*other_series: timeseries.TSContinuous*, *offset: Optional[float] = None*, *inplace: bool = False*) → timeseries.TSContinuous

Append another time series to this one, along the time axis

Parameters

- **other_series** (TSContinuous) – Another time series. Will be tacked on to the end of the called series object. `other_series` must have the same number of channels
- **offset** (*Optional[float]*) – If not None, defines distance between last sample of `self` and first sample of `other_series`. Otherwise the offset will be the median of all timestep sizes of `self.samples`.
- **inplace** (*bool*) – Conduct operation in-place (Default: `False`; create a copy)

Return TSContinuous Time series containing data from `self`, with the other series appended in time

clip(*t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *channels: Union[int, numpy.ndarray, List, Tuple, None] = None*, *include_stop: bool = True*, *sample_limits: bool = True*, *inplace: bool = False*) → timeseries.TSContinuous

Return a TSContinuous which is restricted to given time limits and only contains events of selected channels

Parameters

- **t_start** (*float*) – Time from which on events are returned
- **t_stop** (*float*) – Time until which events are returned
- **channels** (*ArrayLike*) – Channels of which events are returned
- **include_stop** (*bool*) – True -> If there are events with time `t_stop` include them. False -> Exclude these samples. Default: True.
- **sample_limits** (*bool*) – If True, make sure that a sample exists at `t_start` and, if `include_stop` is True, at `t_stop`, as long as not both are None.
- **inplace** (*bool*) – Conduct operation in-place (Default: False; create a copy)

Return TSContinuous clipped_series: New TSContinuous clipped to bounds

contains(*times: Union[int, float, numpy.ndarray, List, Tuple]*) → bool

Does the time series contain the time range specified in the given time trace? Always true for periodic series

Parameters **times** (*ArrayLike*) – Array-like containing time points

Return bool True iff all specified time points are contained within this time series

copy() → timeseries.TimeSeries

Return a deep copy of this time series

Return TimeSeries copy of `self`

delay(*offset: Union[int, float], inplace: bool = False*) → timeseries.TimeSeries

Return a copy of `self` that is delayed by an offset

For delaying self, use the `inplace` argument, or `.times += ...` instead.

Parameters

- **Offset** (*float*) – Time by which to offset this time series
- **inplace** (*bool*) – If `True`, conduct operation in-place (Default: `False`; create a copy)

Return TimeSeries New TimeSeries, delayed

property duration
(float) Duration of TimeSeries

isempty() → bool
Test if this TimeSeries object is empty

Return bool `True` iff the TimeSeries object contains no samples

property max
(float) Maximum value of time series

merge(*other_series: timeseries.TSContinuous*, *remove_duplicates: bool = True*, *inplace: bool = False*) → timeseries.TSContinuous
Merge another time series to this one, by interleaving in time. Maintain each time series' time values and channel IDs.

Parameters

- **other_series** (`TSContinuous`) – time series that is merged to self
- **remove_duplicates** (*Optional[bool]*) – If `True`, time points in `other_series.times` that are also in `self.times` are discarded. Otherwise they are included in the new time trace and come after the corresponding points of self.times.
- **inplace** (*Optional[bool]*) – Conduct operation in-place (Default: `False`; create a copy)

Return TSContinuous The merged time series

property min
(float) Minimum value of time series

property num_channels
(int) Number of channels (dimension of sample vectors) in this TimeSeries object

property num_traces
(int) Synonymous to `num_channels`

plot(*times: Union[int, float, numpy.ndarray, List, Tuple, None] = None*, *target: Union[mpl.axes.Axes, hv.Curve, hv.Overlay, None] = None*, *channels: Union[numpy.ndarray, List, Tuple, int, None] = None*, *stagger: Union[float, int, None] = None*, *skip: Optional[int] = None*, **args*, ***kwargs*)
Visualise a time series on a line plot

Parameters

- **times** (*Optional[ArrayLike]*) – Time base on which to plot. Default: time base of time series
- **target** (*Optional*) – Axes (or other) object to which plot will be added.
- **channels** (*Optional[ArrayLike]*) – Channels of the time series to be plotted.
- **stagger** (*Optional[float]*) – Stagger to use to separate each series when plotting multiple series. (Default: `None`, no stagger)
- **skip** (*Optional[int]*) – Skip several series when plotting multiple series

- **kwargs** (*args,*) – Optional arguments to pass to plotting function

Returns Plot object. Either holoviews Layout, or matplotlib plot

property plotting_backend
(str) Current plotting backend

print(*full: bool = False*, *num_first: int = 4*, *num_last: int = 4*, *limit_shorten: int = 10*)
Print an overview of the time series and its values

Parameters

- **full** (*bool*) – Print all samples of `self`, no matter how long it is
- **num_first** (*int*) – Shortened version of printout contains samples at first `num_first` points in *times*
- **num_last** (*int*) – Shortened version of printout contains samples at last `num_last` points in *times*
- **limit_shorten** (*int*) – Print shortened version of self if it comprises more than `limit_shorten` time points and if `full` is False

resample(*times: Union[int, float, numpy.ndarray, List, Tuple], channels: Union[int, float, numpy.ndarray, List, Tuple, None] = None, inplace: bool = False*) → timeseries.TSContinuous
Return a new time series sampled to the supplied time base

Parameters

- **times** (*ArrayLike*) – T desired time points to resample
- **channels** (*Optional[ArrayLike]*) – Channels to be used. Default: None (use all channels)
- **inplace** (*bool*) – True -> Conduct operation in-place (Default: False; create a copy)

Return TSContinuous Time series resampled to new time base and with desired channels.

property samples
(ArrayLike[float]) Value of time series at sampled times

save(*path: str*, *verbose: bool = False*)
Save this time series as an `npz` file using np.savez

Parameters **path** (*str*) – Path to save file

set_plotting_backend(*backend: Optional[str], verbose: bool = True*)
Set which plotting backend to use with the .plot() method

Parameters

- **backend** (*str*) – Specify a backend to use. Supported: {"holoviews", "matplotlib"}
- **verbose** (*bool*) – If True, print feedback about which backend has been set

property t_start
(float) Start time of time series

property t_stop
(float) Stop time of time series (final sample)

property times
(ArrayLike[float]) Array of sample times

12.2.3 API reference for timeseries.TSEvent

class timeseries.**TSEvent**(*times: Union[numpy.ndarray, List, Tuple] = None*, *channels: Union[int, numpy.ndarray, List, Tuple] = None*, *periodic: bool = False*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *name: str = None*, *num_channels: int = None*)

Bases: *timeseries.TimeSeries*

Represents a discrete time series, composed of binary events (present or absent). This class is primarily used to represent spike trains or event trains to communicate with spiking neuron layers, or to communicate with event-based computing systems. See *Working with time series data* for further explanation and examples.

TSEvent supports multiple channels of event time series encapsulated by a single object, as well as periodic time series.

Examples

Build a series of several random event times

```
>>> times = numpy.cumsum(numpy.random.rand(10))
>>> ts = TSEvent(times)
```

__init__(*times: Union[numpy.ndarray, List, Tuple] = None*, *channels: Union[int, numpy.ndarray, List, Tuple] = None*, *periodic: bool = False*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *name: str = None*, *num_channels: int = None*)

Represent discrete events in time

Parameters

- **times** (*ArrayLike[float]*) – Tx1 vector of event times
- **channels** (*ArrayLike[int]*) – Tx1 vector of event channels (Default: all events are in channel 0)
- **periodic** (*bool*) – Is this a periodic TimeSeries (Default: False; non-periodic)
- **t_start** (*float*) – Explicitly specify the start time of this series. If None, then times[0] is taken to be the start time
- **t_stop** (*float*) – Explicitly specify the stop time of this series. If None, then times[-1] is taken to be the stop time
- **name** (*str*) – Name of the time series (Default: None)
- **num_channels** (*int*) – Total number of channels in the data source. If None, max(channels) is taken to be the total channel number

Attributes

channels	(ArrayLike[int]) Event channel indices.
duration	(float) Duration of TimeSeries
num_channels	(int) The maximum number of channels represented by this *TSEvent*
plotting_backend	(str) Current plotting backend
t_start	(float) Start time of time series
t_stop	(float) Stop time of time series (final sample)
times	(ArrayLike[float]) Array of sample times

Methods

__init__([times, channels, periodic, ...])	Represent discrete events in time
append_c(other_series[, inplace])	Append another time series to `self` along the channels axis
append_t(other_series[, offset, ...])	Append another time series to this one along the time axis
clip([t_start, t_stop, channels, ...])	Return a *TSEvent* which is restricted to given time limits and only contains events of selected channels
copy()	Return a deep copy of this time series
delay(offset[, inplace])	Return a copy of `self` that is delayed by an offset
isempty()	Test if this TimeSeries object is empty
merge(other_series[, delay, ...])	Merge another *TSEvent* into this one so that they may overlap in time
plot([time_limits, target, channels])	Visualise this time series on a scatter plot
print([full, num_first, num_last, limit_shorten])	Print an overview of the time series and its values
raster(dt[, t_start, t_stop, num_timesteps, ...])	Return a rasterized version of the time series data, where each data point represents a time step
remap_channels(channel_map[, inplace])	Renumber channels in the *TSEvent*
save(path[, verbose])	Save this **:py:'TSEvent'** as an `npz` file using `np.savez`
set_plotting_backend(backend[, verbose])	Set which plotting backend to use with the .plot() method
xraster(dt[, t_start, t_stop, ...])	Generator which `yield` s a rasterized time series data, where each data point represents a time step

__init__(*times: Union[numpy.ndarray, List, Tuple] = None*, *channels: Union[int, numpy.ndarray, List, Tuple] = None*, *periodic: bool = False*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *name: str = None*, *num_channels: int = None*)

Represent discrete events in time

Parameters

- **times** (*ArrayLike[float]*) – `Tx1` vector of event times
- **channels** (*ArrayLike[int]*) – `Tx1` vector of event channels (Default: all events are in channel 0)
- **periodic** (*bool*) – Is this a periodic TimeSeries (Default: False; non-periodic)
- **t_start** (*float*) – Explicitly specify the start time of this series. If `None`, then `times[0]` is taken to be the start time
- **t_stop** (*float*) – Explicitly specify the stop time of this series. If `None`, then `times[-1]` is taken to be the stop time
- **name** (*str*) – Name of the time series (Default: None)
- **num_channels** (*int*) – Total number of channels in the data source. If `None`, max(channels) is taken to be the total channel number

_matching_channels(*channels: Union[int, numpy.ndarray, List, Tuple, None] = None*, *event_channels: Union[int, numpy.ndarray, List, Tuple, None] = None*) → numpy.ndarray

Return boolean array of which events match a given channel selection

Parameters **channels** (*ArrayLike[int]*) – Channels of which events are to be indicated `True`. Default: `None`, use all channels

Params ArrayLike[int] event_channels Channel IDs for each event. If not provided (Default: `None`), then use self._channels

Return ArrayLike[bool] A matrix `TxC` indicating which events match the requested channels

_modulo_period(*times: Union[numpy.ndarray, List, Tuple, float, int]*) → Union[numpy.ndarray, List, Tuple, float, int]
_modulo_period - Calculate provided times modulo `self.duration`

append_c(*other_series: timeseries.TSEvent*, *inplace: bool = False*) → timeseries.TSEvent
Append another time series to `self` along the channels axis

The channel IDs in `other_series` are shifted by `self.num_channels`. Event times remain the same.

Parameters

- **other_series** (`TSEvent`) – *TSEvent* or list of *TSEvent* that will be appended to `self`.
- **inplace** (*Optional[bool]*) – Conduct operation in-place (Default: `False`; create a copy)

Return TSEvent *TSEvent* containing data in `self`, with other TS appended along the channels axis

append_t(*other_series: Union[timeseries.TimeSeries, Iterable[timeseries.TimeSeries]], offset: float = 0, remove_duplicates: bool = False, inplace: bool = False*) → timeseries.TSEvent
Append another time series to this one along the time axis

`t_start` from `other_series` is shifted to `self.t_stop + offset`.

Parameters

- **other_series** (`TSEvent`) – *TSEvent* or list of *TSEvent* that will be appended to `self` along the time axis
- **offset** (*Optional[float]*) – Scalar or iterable with at least the same number of elements as `other_series`. If scalar, use same value for all timeseries. Event times from `other_series` will be shifted by `self.t_stop + offset`. Default: 0
- **remove_duplicates** (*Optional[bool]*) – If `True`, duplicate events will be removed from the resulting timeseries. Duplicates can occur if `offset` is negative. Default: `False`, do not remove duplicate events.
- **inplace** (*Optional[bool]*) – If `True`, conduct operation in-place (Default: `False`; return a copy)

Return TSEvent *TSEvent* containing events from `self`, with other TS appended in time

property channels
(ArrayLike[int]) Event channel indices. A `Tx1` vector, where each element `t` corresponds to the event time in `self.times[t]`.

clip(*t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *channels: Union[int, numpy.ndarray, List, Tuple, None] = None*, *include_stop: bool = False*, *remap_channels: bool = False*, *inplace: bool = False*) → timeseries.TSEvent
Return a *TSEvent* which is restricted to given time limits and only contains events of selected channels

If time limits are provided, *t_start* and *t_stop* attributes of the new time series will correspond to those. If *remap_channels* is `True`, channels IDs will be mapped to a continuous sequence of integers starting from 0 (e.g. [1, 3, 6]->[0, 1, 2]). In this case *num_channels* will be set to the number of different channels in `channels`. Otherwise *num_channels* will keep its original values, which is also the case for all other attributes. If `inplace` is True, modify `self` accordingly.

Parameters

- **t_start** (*Optional[float]*) – Time from which on events are returned. Default: *t_start*
- **t_stop** (*Optional[float]*) – Time until which events are returned. Default: *t_stop*
- **channels** (*Optional[ArrayLike[int]]*) – Channels of which events are returned. Default: All channels
- **include_stop** (*Optional[bool]*) – If there are events with time *t_stop*, include them or not. Default: `False`, do not include events at *t_stop*
- **remap_channels** (*Optional[bool]*) – Map channel IDs to continuous sequence starting from 0. Set *num_channels* to largest new ID + 1. Default: `False`, do not remap channels
- **inplace** (*Optional[bool]*) – Iff `True`, the operation is performed in place (Default: False)

Return TSEvent *TSEvent* containing events from the requested channels

copy() → timeseries.TimeSeries
Return a deep copy of this time series

Return TimeSeries copy of `self`

delay(*offset: Union[int, float], inplace: bool = False*) → timeseries.TimeSeries
Return a copy of `self` that is delayed by an offset

For delaying self, use the `inplace` argument, or `.times += ...` instead.

Parameters

- **Offset** (*float*) – Time by which to offset this time series
- **inplace** (*bool*) – If `True`, conduct operation in-place (Default: `False`; create a copy)

Return TimeSeries New TimeSeries, delayed

property duration
(float) Duration of TimeSeries

isempty() → bool
Test if this TimeSeries object is empty

Return bool `True` iff the TimeSeries object contains no samples

merge(*other_series: Union[timeseries.TimeSeries, Iterable[timeseries.TimeSeries]], delay: Union[float, Iterable[float]] = 0, remove_duplicates: bool = False, inplace: bool = False*) → timeseries.TSEvent
Merge another *TSEvent* into this one so that they may overlap in time

Parameters

- **other_series** (`TSEvent`) – *TSEvent* or list of *TSEvent* to merge into `self`
- **delay** (*Optional[float]*) – Scalar or iterable with at least the number of elements as other_series. If scalar, use same value for all timeseries. Delay `other_series` series by this value before merging.
- **remove_duplicates** (*Optional[bool]*) – If `True`, remove duplicate events in resulting timeseries. Default: `False`, do not remove duplicates.

- **inplace** (*Optional[bool]*) – If `True`, operation will be performed in place (Default: `False`, return a copy)

Return TSEvent `self` with new samples included

property num_channels
(int) The maximum number of channels represented by this *TSEvent*

plot (*time_limits: Optional[Tuple[Optional[float], Optional[float]]] = None, target: Union[mpl.axes.Axes, hv.Scatter, hv.Overlay, None] = None, channels: Union[numpy.ndarray, List, Tuple, int, None] = None, *args, **kwargs*)
Visualise this time series on a scatter plot

Parameters

- **float] time_limits** (*Optional[float,]*) – Tuple with times between which to plot. Default: plot all times
- **target** (*Optional[axis]*) – Object to which plot will be added. Default: new plot
- **channels** (*ArrayLike[int]*) – Channels that are to be plotted. Default: plot all channels
- **kwargs** (*args,*) – Optional arguments to pass to plotting function

Returns Plot object. Either holoviews Layout, or matplotlib plot

property plotting_backend
(str) Current plotting backend

print (*full: bool = False, num_first: int = 4, num_last: int = 4, limit_shorten: int = 10*)
Print an overview of the time series and its values

Parameters

- **full** (*bool*) – Print all samples of `self`, no matter how long it is. Default: `False`
- **limit_shorten** (*int*) – Print shortened version of `self` if it comprises more than `limit_shorten` time points and `full` is `False`. Default: 4
- **num_first** (*int*) – Shortened version of printout contains samples at first `num_first` points in `self.times`. Default: 4
- **num_last** (*int*) – Shortened version of printout contains samples at last `num_last` points in `self.times`. Default: 4

raster (*dt: float, t_start: float = None, t_stop: float = None, num_timesteps: int = None, channels: numpy.ndarray = None, add_events: bool = False*) → numpy.ndarray
Return a rasterized version of the time series data, where each data point represents a time step

Events are represented in a boolean matrix, where the first axis corresponds to time, the second axis to the channel. Events that happen between time steps are projected to the preceding step. If two events happen during one time step within a single channel, they are counted as one, unless `add_events` is `True`.

Parameters

- **dt** (*float*) – Duration of single time step in raster
- **t_start** (*Optional[float]*) – Time where to start raster. Default: None (use `self.t_start`)
- **t_stop** (*Optional[float]*) – Time where to stop raster. This time point is not included in the raster. Default: `None` (use `self.t_stop`. If `num_timesteps` is provided, `t_stop` is ignored.

- **num_timesteps** (*Optional[int]*) – Specify number of time steps directly, instead of providing `t_stop`. Default: `None` (use `t_start`, `t_stop` and `dt` to determine raster size)
- **channels** (*Optional[ArrayLike[int]]*) – Channels from which data is to be used. Default: `None` (use all channels)
- **add_events** (*Optional[bool]*) – If `True`, return an integer raster containing number of events for each time step and channel. Default: `False`, merge simultaneous events in a single channel, and return a boolean raster

Return ArrayLike event_raster - Boolean matrix with `True` indicating presence of events for each time step and channel. If `add_events == True`, the raster consists of integers indicating the number of events per time step and channel. First axis corresponds to time, second axis to channel.

remap_channels(*channel_map: Union[numpy.ndarray, List, Tuple], inplace: bool = False*) → timeseries.TSEvent

Renumber channels in the *TSEvent*

Maps channels 0..‘‘self.num_channels-1‘‘ to the channels in `channel_map`.

Parameters

- **channel_map** (*ArrayLike[int]*) – List of channels that existing channels should be mapped to, in order.. Must be of size `self.num_channels`.
- **inplace** (*bool*) – Specify whether operation should be performed in place (Default: `False`, a copy is returned)

save(*path: str*, *verbose: bool = False*)

Save this **:py:‘TSEvent‘** as an `npz` file using `np.savez`

Parameters **path** (*str*) – File path to save data

set_plotting_backend(*backend: Optional[str], verbose: bool = True*)

Set which plotting backend to use with the .plot() method

Parameters

- **backend** (*str*) – Specify a backend to use. Supported: {“holoviews”, “matplotlib”}
- **verbose** (*bool*) – If True, print feedback about which backend has been set

property t_start

(float) Start time of time series

property t_stop

(float) Stop time of time series (final sample)

property times

(ArrayLike[float]) Array of sample times

xraster(*dt: float*, *t_start: Optional[float] = None*, *t_stop: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *channels: Optional[numpy.ndarray] = None*) → numpy.ndarray

Generator which `yield`s a rasterized time series data, where each data point represents a time step

Events are represented in a boolean matrix, where the first axis corresponds to time, the second axis to the channel. Events that happen between time steps are projected to the preceding one. If two events happen during one time step within a single channel, they are counted as one.

Parameters

- **dt** (*float*) – Duration of single time step in raster

- **t_start** (*Optional[float]*) – Time where to start raster. Default: `None` (use `self.t_start`)
- **t_stop** (*Optional[float]*) – Time where to stop raster. This time point is not included in the raster. Default: `None` (use `self.t_stop`. If `num_timesteps` is provided, `t_stop` is ignored.
- **num_timesteps** (*Optional[int]*) – Specify number of time steps directly, instead of providing `t_stop`. Default: `None` (use `t_start`, `t_stop` and `dt` to determine raster size.
- **channels** (*Optional[ArrayLike[int]]*) – Channels from which data is to be used. Default: `None` (use all channels)

Yields ArrayLike event_raster - Boolean matrix with `True` indicating presence of events for each time step and channel. If `add_events == True`, the raster consists of integers indicating the number of events per time step and channel. First axis corresponds to time, second axis to channel.

12.3 Utility modules

/reference/weights.rst provides several useful functions for generating network weights.

/reference/utils.rst provides several useful utility functions.

12.4 `Layer` subclasses

See also:

Types of Layer available in Rockpool, *Building and simulating a reservoir network* and other tutorials.

layers.RecRateEuler(weights[, bias, tau, ...])	A standard recurrent non-spiking layer of dynamical neurons
layers.FFRateEuler(weights[, dt, name, ...])	Feedforward layer consisting of rate-based neurons
layers.PassThrough(weights[, dt, noise_std, ...])	Feed-forward layer with neuron states directly corresponding to input with an optional delay
layers.FFIAFBrian(weights[, bias, dt, ...])	*DEPRECATED* A spiking feedforward layer with current inputs and spiking outputs
layers.FFIAFSpkInBrian(weights[, bias, dt, ...])	*DEPRECATED* Spiking feedforward layer with spiking inputs and outputs
layers.RecIAFBrian([weights, bias, dt, ...])	*DEPRECATED* A spiking recurrent layer with current inputs and spiking outputs, using a Brian2 backend
layers.RecIAFSpkInBrian(weights_in, weights_rec)	*DEPRECATED* Spiking recurrent layer with spiking in- and outputs, and a Brian2 backend
layers.PassThroughEvents(weights[, dt, ...])	Pass through events by routing to different channels
layers.FFExpSynBrian([weights, dt, ...])	*DEPRECATED* Define an exponential synapse layer (spiking input), with a Brian2 backend
layers.FFExpSyn(weights[, bias, dt, ...])	Define an exponential synapse layer with spiking inputs and current outputs
layers.RecLIFJax(w_recurrent, tau_mem, tau_syn)	Recurrent spiking neuron layer (LIF), spiking input and spiking output.

Continued on next page

Table 18 – continued from previous page

layers.RecLIFCurrentInJax(w_recurrent, ...)	Recurrent spiking neuron layer (LIF), current injection input and spiking output.
layers.RecLIFJax_IO(w_in, w_recurrent, ...)	Recurrent spiking neuron layer (LIF), spiking input and weighted surrogate output.
layers.RecLIFCurrentInJax_IO(w_in, ...[, ...])	Recurrent spiking neuron layer (LIF), weighted current input and weighted surrogate output.
layers.FFCLIAF(weights[, bias, v_thresh, ...])	Feedforward layer of integrate and fire neurons with constant leak
layers.RecCLIAF(weights_in, weights_rec[, ...])	Recurrent layer of integrate and fire neurons with constant leak
layers.CLIAF(weights_in[, bias, v_thresh, ...])	Abstract layer class of integrate and fire neurons with constant leak
layers.SoftMaxLayer([weights, thresh, dt, name])	A spiking SoftMax layer with spiking inputs and outputs, and constant leak
layers.RecDIAF(weights_in, weights_rec[, ...])	Define a spiking recurrent layer based on quantized digital IAF neurons
layers.RecFSSpikeEulerBT([weights_fast, ...])	Implement a spiking reservoir with tight E/I balance.
layers.FFUpDown(weights[, repeat_output, ...])	Define a spiking feedforward layer to convert analogue inputs to up and down channels
layers.FFExpSynTorch([weights, bias, dt, ...])	Define an exponential synapse layer (spiking input, pytorch as backend)
layers.FFIAFTorch(weights[, bias, dt, ...])	Define a spiking feedforward layer with spiking outputs, with a PyTorch backend
layers.FFIAFRefrTorch(weights[, bias, dt, ...])	A spiking feedforward layer with spiking outputs and refractoriness
layers.FFIAFSpkInTorch(weights[, bias, dt, ...])	Spiking feedforward layer with spiking in- and outputs
layers.FFIAFSpkInRefrTorch(weights[, bias, ...])	Spiking feedforward layer with spiking in- and outputs and refractoriness, using a PyTorch backend
layers.RecIAFTorch(weights[, bias, dt, ...])	A spiking recurrent layer with input currents and spiking outputs
layers.RecIAFRefrTorch(weights[, bias, dt, ...])	A spiking recurrent layer with current inputs, spiking outputs and refractoriness.
layers.RecIAFSpkInTorch(weights_in, weights_rec)	A spiking recurrent layer with spiking in- and outputs
layers.RecIAFSpkInRefrTorch(weights_in, ...)	A spiking recurrent layer with spiking in- and outputs and refractoriness, and a PyTorch backend
layers.RecIAFSpkInRefrCLTorch(weights_in, ...)	A recurrent spiking layer with constant leak.
layers.FFIAFNest	
layers.RecIAFSpkInNest	
layers.RecAEIFSpkInNest	
layers.RecDynapSE(weights_in, weights_rec[, ...])	Recurrent spiking layer implemented with a DynapSE backend.
layers.VirtualDynapse	
layers.RecRateEulerJax(w_in, w_recurrent, ...)	JAX-backed firing-rate recurrent layer
layers.ForceRateEulerJax(w_in, w_out, tau, bias)	Implements a pseudo recurrent reservoir, for use in reservoir transfer

12.4.1 API reference for layers.RecRateEuler

class layers.**RecRateEuler**(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.0*, *tau: numpy.ndarray = 1.0*, *activation_func: Callable[numpy.ndarray, numpy.ndarray] = CPUDispatcher(<function re_lu>)*, *dt: float = None*, *noise_std: float = 0.0*, *name: str = None*)

Bases: rockpool.layers.layer.Layer

A standard recurrent non-spiking layer of dynamical neurons

RecRateEuler implements a very standard recurrent layer of dynamical neurons, which by default have linear-threshold (or "rectified-linear", or ReLU) transfer function. The layer is backed by a simple forward-Euler solver with a fixed time step.

The neurons in the layer implement the dynamical system

$$\tau \cdot \dot{x} + x = WH(x + b) + I(t) + \sigma \cdot \zeta(t)$$

where x is the Nx1 vector of internal states of each neuron; $\dot{x}$ is the derivative of these states with respect to time; τ is the vector of time constants for each neurons; W is the [NxN] recurrent weight matrix for this layer; b is the Nx1 vector of neuron biases for this layer; $I(t)$ is the input injected into each neuron in this layer at time t; $\sigma \cdot \zeta(t)$ is a white noise process with standard deviation σ at each time step. $H(x)$ is the neuron transfer function, which by default is the linear threshold function

$$H(x) = max(0, x)$$

See also:

The tutorial *Building and simulating a reservoir network* demonstrates using this layer.

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.0*, *tau: numpy.ndarray = 1.0*, *activation_func: Callable[numpy.ndarray, numpy.ndarray] = CPUDispatcher(<function re_lu>)*, *dt: float = None*, *noise_std: float = 0.0*, *name: str = None*)

Implement a recurrent layer with non-spiking firing rate neurons, using a forward-Euler solver

Parameters

- **weights** (*ndarray*) – (NxN) matrix of recurrent weights
- **bias** (*Optional[ArrrayLike[float]]*) –

(N) vector (or scalar) of bias currents. Default: 0.0

- **tau** (*Optional[ArrrayLike[float]]*) –

(N) vector (or scalar) of neuron time constants. Default: 1.0

- **float] activation_func** (*Callable[[float],*) – Activation function for each neuron, with signature (x) -> f(x). Default: re_lu
- **dt** (*Optional[float]*) – Time step for integration (Euler method). Default: None, which results in taking a minimum time step of min(tau) / 10.0 for numerical stability.
- **noise_std** (*Optional[float]*) – Std. Dev. of state noise injected at each time step. Default: 0.0, no noise
- **name** (*Optional[str]*) – Name of this layer. Default: None

Attributes

activation_func	(Callable) Activation function for this layer
bias	(ndarray) (N) Vector of bias values for the neurons in this layer :return:
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau	(ndarray) (N) Vector of time constants for the neurons in this layer :return:
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, bias, tau, ...])	Implement a recurrent layer with non-spiking firing rate neurons, using a forward-Euler solver
evolve([ts_input, duration, num_timesteps, ...])	Evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
stream(duration, dt[, verbose])	Stream data through this layer
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.0*, *tau: numpy.ndarray = 1.0*, *activation_func: Callable[numpy.ndarray, numpy.ndarray] = CPUDispatcher(<function re_lu>)*, *dt: float = None*, *noise_std: float = 0.0*, *name: str = None*)

Implement a recurrent layer with non-spiking firing rate neurons, using a forward-Euler solver

Parameters

- **weights** (*ndarray*) – (NxN) matrix of recurrent weights
- **bias** (*Optional[ArrrayLike[float]]*) –

(N) vector (or scalar) of bias currents. Default: 0.0

- **tau** (*Optional[ArrrayLike[float]]*) –

(N) vector (or scalar) of neuron time constants. Default: 1.0

- **float] activation_func** (*Callable[[float],*) – Activation function for each neuron, with signature (x) -> f(x). Default: `re_lu`
- **dt** (*Optional[float]*) – Time step for integration (Euler method). Default: `None`, which results in taking a minimum time step of `min(tau) / 10.0` for numerical stability.
- **noise_std** (*Optional[float]*) – Std. Dev. of state noise injected at each time step. Default: 0.0, no noise
- **name** (*Optional[str]*) – Name of this layer. Default: `None`

_check_input_dims (*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size (*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace (*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)
Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property activation_func
(Callable) Activation function for this layer

This function must have the signature Callable[[ndarray], ndarray]

property bias
(ndarray) (N) Vector of bias values for the neurons in this layer :return:

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous
Evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series to use during evolution. Default: `None`, do not inject any input
- **duration** (*Optional[float]*) – Desired evolution time in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` will determine evolution duration
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` will determine evolution duration
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSContinuous output time series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored

- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

stream(*duration: float*, *dt: float*, *verbose: bool = False*) → Tuple[float, List[float]]
Stream data through this layer

Parameters

- **duration** (*float*) – Total duration for which to handle streaming

- **dt** (*float*) – Streaming time step
- **verbose** (*bool*) – Display feedback

Yield (float, ndarray) (t, state)

Return (float, ndarray) Final output (t, state)

property t
(float) The current evolution time of this layer

property tau
(ndarray) (N) Vector of time constants for the neurons in this layer :return:

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

Return dict Dictionary of parameters to use when reconstructing this layer

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.2 API reference for layers.FFRateEuler

class layers.**FFRateEuler**(*weights: numpy.ndarray, dt: Optional[float] = None, name: Optional[str] = None, noise_std: Optional[float] = 0.0, activation_func: Optional[Callable[numpy.ndarray, numpy.ndarray]] = CPUDispatcher(<function re_lu>), tau: Union[float, numpy.ndarray, None] = 10.0, gain: Union[float, numpy.ndarray, None] = 1.0, bias: Union[float, numpy.ndarray, None] = 0.0*)
Bases: `rockpool.layers.training.gpl.rr_trained_layer.RRTrainedLayer`

Feedforward layer consisting of rate-based neurons

FFRateEuler is a simple feed-forward layer of dynamical neurons, backed with a forward-Euler solver with a fixed time step. The neurons in this layer implement the dynamics

$$\tau \cdot \dot{x} + x = g \cdot WI(t) + \sigma \cdot \zeta(t)$$

where x is the Nx1 vector of internal states of neurons in the layer; $\dot{x}$ is the derivative of those staes with respect to time; τ is the vector of time constants of the neurons in the layer; $I(t)$ is the instantaneous input injected into each neuron at time t; W is the MxN matrix of weights connecting the input to the neurons in the layer; and $\sigma \cdot \zeta(t)$ is a white noise process with standard deviation σ.

The output of the layer is given by

$$o = H(x + b)$$

where $H(x)$ is the neuron transfer function, which by default is the linear-threshold (or "rectified linear" or ReLU) function $H(x) = max(0, x)$; b is the Nx1 vector of bias values for this layer; and g is the Nx1 vector of gain parameters for the neurons in this layer.

Training

FFRateEuler supports weight training with linear or ridge regression, using the *train* method. To use this facility, use the *train* method instead of the *evolve* method, calling *train* in turn over multiple batches:

```
lyr = FFRateEuler(...)

# - Loop over batches and train
```

(continues on next page)

(continued from previous page)

```
is_first = True
is_last = False

for (input_batch_ts, target_batch_ts) in batches[:-1]:
    lyr.train(target_batch_tsm, input_batch_ts, is_first, is_last)
    is_first = False

# - Finalise training for last batch
is_last = True
(input_batch_ts, target_batch_ts) = batches[-1]
lyr.train(target_batch_ts, input_batch_ts, is_first, is_last)
```

__init__(*weights: numpy.ndarray, dt: Optional[float] = None, name: Optional[str] = None, noise_std: Optional[float] = 0.0, activation_func: Optional[Callable[numpy.ndarray, numpy.ndarray]] = CPUDispatcher(<function re_lu>), tau: Union[float, numpy.ndarray, None] = 10.0, gain: Union[float, numpy.ndarray, None] = 1.0, bias: Union[float, numpy.ndarray, None] = 0.0*)

Implement a feed-forward non-spiking neuron layer, with an Euler method solver

Parameters

- **weights** (*ndarray*) – [MxN] Weight matrix
- **dt** (*Optional[float]*) – Time step for Euler solver, in seconds. Default: `None`, which will use `min(tau) / 10` as the time step, for numerical stability
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: 0.0, no noise
- **float] activation_func** (*Optional[Callable[[float],]*) – Callable a = f(x) Neuron activation function. Default: ReLU
- **tau** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of neuron time constants in seconds. Default: 10.0
- **gain** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of gain factors. Default: 1.0, unitary gain
- **bias** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of bias currents. Default: 0.0

Attributes

activation	(ArrayLike[float]) The activation of this layer, after the activation function
activation_func	(Callable[[ndarray], ndarray) Activation function for the neurons in this layer
alpha	(ndarray) (N) Vector *tau* / *dt* for the neurons in this layer
bias	(ArrayLike[float]) (N) Vector of bias parameters for this layer
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
gain	(ArrayLike[float]) (N) Vector of gain parameters for this layer

Continued on next page

Table 21 – continued from previous page

input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau	(ArayLike[float]) (N) Vector of time constants for the neurons in this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, dt, name, noise_std, ...])	Implement a feed-forward non-spiking neuron layer, with an Euler method solver
evolve([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
stream(duration, dt[, verbose])	Stream data through this layer
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer
train(ts_target, ts_input, is_first, is_last)	Wrapper to standardize training syntax across layers.
train_rr(ts_target[, ts_input, regularize, ...])	Train this layer with ridge regression over one of possibly many batches.

__init__(*weights: numpy.ndarray*, *dt: Optional[float] = None*, *name: Optional[str] = None*, *noise_std: Optional[float] = 0.0*, *activation_func: Optional[Callable[numpy.ndarray, numpy.ndarray]] = CPUDispatcher(<function re_lu>)*, *tau: Union[float, numpy.ndarray, None] = 10.0*, *gain: Union[float, numpy.ndarray, None] = 1.0*, *bias: Union[float, numpy.ndarray, None] = 0.0*)

Implement a feed-forward non-spiking neuron layer, with an Euler method solver

Parameters

- **weights** (*ndarray*) – [MxN] Weight matrix
- **dt** (*Optional[float]*) – Time step for Euler solver, in seconds. Default: `None`, which will use `min(tau) / 10` as the time step, for numerical stability
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: 0.0, no noise
- **float] activation_func** (*Optional[Callable[[float],*) – Callable a = f(x) Neuron activation function. Default: ReLU
- **tau** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of neuron time constants in seconds. Default: 10.0
- **gain** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of gain factors. Default: 1.0, unitary gain
- **bias** (*Optional[ArrayLike[float]]*) – [Nx1] Vector of bias currents. Default: 0.0

_batch_update(*inp: numpy.ndarray*, *target: numpy.ndarray*, *reset: bool*, *train_biases: bool*, *standardize: bool*, *update_weights: bool*, *return_training_progress: bool*) → Dict

Train with the already processed input and target data of the current batch. Update layer weights and biases if requested. Provide information on training state if requested.

Parameters

- **inp** (*np.ndarray*) – 2D-array (num_samples x num_features) of input data.
- **target** (*np.ndarray*) – 2D-array (num_samples x 'self.size') of target data.
- **reset** (*bool*) – If 'True', internal variables will be reset at the end.
- **train_bises** (*bool*) – Should biases be trained or only weights?
- **standardize** (*bool*) – Has input data been z-score standardized?
- **update_weights** (*bool*) – Set 'True' to update layer weights and biases.
- **return_training_progress** (*bool*) – Return intermediate training data (e.g. xtx, xty,...)

Return dict Dict with information on training progress, depending on values of other function arguments.

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_correct_param_shape(*v*) → numpy.ndarray

Convert an argument to a 1D-np.ndarray and verify that the dimensions match `self.size`

Parameters **v** (*float*) – Scalar or array-like that is to be converted

Returns v as 1D-np.ndarray, possibly expanded to `self.size`

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time

- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_training_data (*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*)

Check and rasterize input and target data for this batch

Parameters

- **ts_target** (*TSContinuous*) – Target time series for this batch
- **None]] ts_input** (*Optional[Union[TSContinuous,*) – Input time series for this batch. Default: `None`
- **is_first** (*Optional[bool]*) – Set to `True` if this is the first batch in training. Default: `True`
- **is_last** (*Optional[bool]*) – Set to `True` if this is the last batch in training. Default: `False`

Returns (inp, target, time_base) inp np.ndarray: Rasterized input time series for this batch [T, M] target np.ndarray: Rasterized target time series for this batch [T, O] time_base np.ndarray: Time base for `inp` and `target` [T,]

property activation

(ArrayLike[float]) The activation of this layer, after the activation function

property activation_func

(Callable[[ndarray], ndarray) Activation function for the neurons in this layer

property alpha

(ndarray) (N) Vector *tau* / *dt* for the neurons in this layer

property bias
(ArrayLike[float]) (N) Vector of bias parameters for this layer

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous
Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series. Default: `None`, no stimulus is provided
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time
- **Optional[bool]verbose** – Currently no effect, just for conformity

Return TSContinuous Output time series

property gain
(ArrayLike[float]) (N) Vector of gain parameters for this layer

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

stream(*duration: float*, *dt: float*, *verbose: Optional[bool] = False*) → Tuple[float, List[float]]
Stream data through this layer

Parameters

- **duration** (*float*) – Total duration for which to handle streaming
- **dt** (*float*) – Streaming time step
- **verbose** (*Optional[bool]*) – Display feedback. Default: `False`, don't display feedback

Yield (float, ndarray) (t, state)

Return (float, ndarray) Final output (t, state)

property t
(float) The current evolution time of this layer

property tau
(ArayLike[float]) (N) Vector of time constants for the neurons in this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

Return dict Dictionary of parameters to use when reconstructing this layer

train(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSContinuous*, *is_first: bool*, *is_last: bool*, *method: str = 'rr'*, ***kwargs*)
Wrapper to standardize training syntax across layers. Use specified training method to train layer for current batch.

Parameters

- **ts_target** – Target time series for current batch.
- **ts_input** – Input to the layer during the current batch.
- **is_first** – Set `True` to indicate that this batch is the first in training procedure.
- **is_last** – Set `True` to indicate that this batch is the last in training procedure.
- **method** – String indicating which training method to choose. Currently only ridge regression (`'rr'`) is supported.
- **kwargs** – will be passed on to corresponding training method.

train_rr(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Union[rockpool.timeseries.TSEvent, rockpool.timeseries.TSContinuous, None] = None*, *regularize: Optional[float] = 0*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*, *train_biases: Optional[bool] = True*, *calc_intermediate_results: Optional[bool] = False*, *return_training_progress: Optional[bool] = True*, *return_trained_output: Optional[bool] = False*, *fisher_relabelling: Optional[bool] = False*, *standardize: Optional[bool] = False*) → Optional[Dict]
Train this layer with ridge regression over one of possibly many batches. Use Kahan summation to reduce rounding errors when adding data to existing matrices from previous batches.

Parameters

- **ts_target** (*TSContinuous*) – Target signal for current batch
- **ts_input** (*Optional[TimeSeries]*) – Input to layer for current batch. Default: `None`, no input for this batch
- **regularize** (*Optional[float]*) – Regularization parameter for ridge regression. Default: `0`, no regularization
- **is_first** (*Optional[bool]*) – Set to `True` if current batch is the first in training. Default: `True`, initialise training with this batch as the first batch
- **is_last** (*Optional[bool]*) – Set to `True` if current batch is the last in training. This has the same effect as if data from both trainings were presented at once.
- **train_biases** (*Optional[bool]*) – If `True`, train biases as if they were weights. Otherwise present biases will be ignored in training and not be changed. Default: `True`, train biases as well as weights

- **calc_intermediate_results** (*Optional[bool]*) – If `True`, calculates the intermediate weights not in the final batch. Default: `False`, do not compute intermediate weights
- **return_training_progress** (*Optional[bool]*) – If `True`, return dict of current training variables for each batch. Default: `True`, return training progress
- **return_trained_output** (*Optional[bool]*) – If `True`, return the result of evolving the layer with the trained weights in the output dict. Default: `False`, do not return the trained output
- **fisher_relabelling** (*Optional[bool]*) – If `True`, relabel target data such that the training algorithm is equivalent to Fisher discriminant analysis. Default: `False`, use standard ridge / linear regression
- **standardize** (*Optional[bool]*) – Train with z-score standardized data, based on means and standard deviations from first batch. Default: `False`, do not standardize data

Returns If `return_training_progress` is `True`, return a dict with current training variables (xtx, xty, kahan_comp_xtx, kahan_comp_xty). Weights and biases are returned if `is_last` is `True` or if `calc_intermediate_results` is `True`. If `return_trained_output` is `True`, the dict contains the output of evolving the layer with the newly trained weights.

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.3 API reference for layers.PassThrough

class layers.**PassThrough**(*weights: numpy.ndarray, dt: float = 1.0, noise_std: float = 0.0, bias: Union[float, numpy.ndarray] = 0.0, delay: float = 0.0, name: str = None*)
Bases: `rockpool.layers.gpl.rate.FFRateEuler`

Feed-forward layer with neuron states directly corresponding to input with an optional delay

__init__(*weights: numpy.ndarray, dt: float = 1.0, noise_std: float = 0.0, bias: Union[float, numpy.ndarray] = 0.0, delay: float = 0.0, name: str = None*)
Implement a feed-forward layer that simply passes input (possibly delayed)

Parameters

- **weights** (*ndarray*) – [MxN] Weight matrix
- **dt** (*Optional[float]*) – Time step for Euler solver, in seconds. Default: 1.0
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: 0.0, no noise
- **bias** (*Optional[ndarray]*) – [Nx1] Vector of bias currents. Default: 0.0, no bias
- **delay** (*Optional[float]*) – Delay between input and output, in seconds. Default: 0.0, no delay
- **name** (*Optional[str]*) – Name of this layer. Default: None

Attributes

activation	(ArrayLike[float]) The activation of this layer, after the activation function

Continued on next page

Table 23 – continued from previous page

activation_func	(Callable[[ndarray], ndarray) Activation function for the neurons in this layer
alpha	(ndarray) (N) Vector *tau* / *dt* for the neurons in this layer
bias	(ArrayLike[float]) (N) Vector of bias parameters for this layer
buffer	(ndarray) The internal buffer of this layer.
class_name	(str) Class name of `self`
delay	(float) The delay imposed by this layer, in seconds
delay_steps	(int) The delay imposed by this layer, in units of *dt*
dt	(float) Simulation time step of this layer
gain	(ArrayLike[float]) (N) Vector of gain parameters for this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau	(ArayLike[float]) (N) Vector of time constants for the neurons in this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, dt, noise_std, bias, ...])	Implement a feed-forward layer that simply passes input (possibly delayed)
evolve([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
print_buffer(**kwargs)	Display the internal buffer of this layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal state and time stamp of this layer
reset_buffer()	Reset the internal buffer of this layer
reset_state()	Reset the internal state and buffer of this layer to zero
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file

Continued on next page

Table 24 – continued from previous page

save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
stream(duration, dt[, verbose])	Stream data through this layer
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer
train(ts_target, ts_input, is_first, is_last)	Wrapper to standardize training syntax across layers.
train_rr(ts_target[, ts_input, regularize, ...])	Train this layer with ridge regression over one of possibly many batches.

__init__(*weights: numpy.ndarray*, *dt: float = 1.0*, *noise_std: float = 0.0*, *bias: Union[float, numpy.ndarray] = 0.0*, *delay: float = 0.0*, *name: str = None*)
Implement a feed-forward layer that simply passes input (possibly delayed)

Parameters

- **weights** (*ndarray*) – [MxN] Weight matrix
- **dt** (*Optional[float]*) – Time step for Euler solver, in seconds. Default: 1.0
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: 0.0, no noise
- **bias** (*Optional[ndarray]*) – [Nx1] Vector of bias currents. Default: 0.0, no bias
- **delay** (*Optional[float]*) – Delay between input and output, in seconds. Default: 0.0, no delay
- **name** (*Optional[str]*) – Name of this layer. Default: None

_batch_update(*inp: numpy.ndarray*, *target: numpy.ndarray*, *reset: bool*, *train_biases: bool*, *standardize: bool*, *update_weights: bool*, *return_training_progress: bool*) → Dict
Train with the already processed input and target data of the current batch. Update layer weights and biases if requested. Provide information on training state if requested.

Parameters

- **inp** (*np.ndarray*) – 2D-array (num_samples x num_features) of input data.
- **target** (*np.ndarray*) – 2D-array (num_samples x 'self.size') of target data.
- **reset** (*bool*) – If 'True', internal variables will be reset at the end.
- **train_bises** (*bool*) – Should biases be trained or only weights?
- **standardize** (*bool*) – Has input data been z-score standardized?
- **update_weights** (*bool*) – Set 'True' to update layer weights and biases.
- **return_training_progress** (*bool*) – Return intermediate training data (e.g. xtx, xty,...)

Return dict Dict with information on training progress, depending on values of other function arguments.

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_correct_param_shape(*v*) → numpy.ndarray
Convert an argument to a 1D-np.ndarray and verify that the dimensions match `self.size`

Parameters **v** (*float*) – Scalar or array-like that is to be converted

Returns v as 1D-np.ndarray, possibly expanded to `self.size`

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of .dt. If not provided, then duration or the duration of ts_input will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of .dt

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a TSEvent time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of .dt. If not provided, then either duration or the duration of ts_input will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of .dt

_prepare_training_data (*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*)

Check and rasterize input and target data for this batch

Parameters

- **ts_target** (TSContinuous) – Target time series for this batch
- **None]] ts_input** (*Optional[Union[TSContinuous,*) – Input time series for this batch. Default: None
- **is_first** (*Optional[bool]*) – Set to True if this is the first batch in training. Default: True
- **is_last** (*Optional[bool]*) – Set to True if this is the last batch in training. Default: False

Returns (inp, target, time_base) inp np.ndarray: Rasterized input time series for this batch [T, M] target np.ndarray: Rasterized target time series for this batch [T, O] time_base np.ndarray: Time base for inp and target [T,]

property activation

(ArrayLike[float]) The activation of this layer, after the activation function

property activation_func
(Callable[[ndarray], ndarray) Activation function for the neurons in this layer

property alpha
(ndarray) (N) Vector *tau* / *dt* for the neurons in this layer

property bias
(ArrayLike[float]) (N) Vector of bias parameters for this layer

property buffer
(ndarray) The internal buffer of this layer.

property class_name
(str) Class name of `self`

property delay
(float) The delay imposed by this layer, in seconds

property delay_steps
(int) The delay imposed by this layer, in units of *dt*

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous
Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` will be used for the evolution duration

:param Optional[int] num_timesteps Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` will be used for the evolution duration :param Optional[bool] verbose: Currently has no effect

Return TSContinuous Output time series

property gain
(ArrayLike[float]) (N) Vector of gain parameters for this layer

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

print_buffer (***kwargs*)
Display the internal buffer of this layer

Parameters **kwargs** – Optional arguments passed to the *TSContinuous.print* method

randomize_state ()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all ()
Reset both the internal state and time stamp of this layer

reset_buffer ()
Reset the internal buffer of this layer

This method will wipe the internal buffer to zeros.

reset_state ()
Reset the internal state and buffer of this layer to zero

reset_time ()
Reset the internal clock of this layer to 0

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

stream(*duration: float*, *dt: float*, *verbose: Optional[bool] = False*) → Tuple[float, List[float]]
Stream data through this layer

Parameters

- **duration** (*float*) – Total duration for which to handle streaming
- **dt** (*float*) – Streaming time step
- **verbose** (*Optional[bool]*) – Display feedback. Default: `False`, don't display feedback

Yield (float, ndarray) (t, state)

Return (float, ndarray) Final output (t, state)

property t
(float) The current evolution time of this layer

property tau
(ArayLike[float]) (N) Vector of time constants for the neurons in this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

Return dict A dictionary containing the parameters of this layer

train(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSContinuous*, *is_first: bool*, *is_last: bool*, *method: str = 'rr'*, ***kwargs*)
Wrapper to standardize training syntax across layers. Use specified training method to train layer for current batch.

Parameters

- **ts_target** – Target time series for current batch.
- **ts_input** – Input to the layer during the current batch.
- **is_first** – Set `True` to indicate that this batch is the first in training procedure.
- **is_last** – Set `True` to indicate that this batch is the last in training procedure.
- **method** – String indicating which training method to choose. Currently only ridge regression (`'rr'`) is supported.
- **kwargs** – will be passed on to corresponding training method.

train_rr(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Union[rockpool.timeseries.TSEvent, rockpool.timeseries.TSContinuous, None] = None*, *regularize: Optional[float] = 0*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*, *train_biases: Optional[bool] = True*, *calc_intermediate_results: Optional[bool] = False*, *return_training_progress: Optional[bool] = True*, *return_trained_output: Optional[bool] = False*, *fisher_relabelling: Optional[bool] = False*, *standardize: Optional[bool] = False*) → Optional[Dict]
Train this layer with ridge regression over one of possibly many batches. Use Kahan summation to reduce

rounding errors when adding data to existing matrices from previous batches.

Parameters

- **ts_target** (*TSContinuous*) – Target signal for current batch
- **ts_input** (*Optional[TimeSeries]*) – Input to layer for current batch. Default: `None`, no input for this batch
- **regularize** (*Optional[float]*) – Regularization parameter for ridge regression. Default: `0`, no regularization
- **is_first** (*Optional[bool]*) – Set to `True` if current batch is the first in training. Default: `True`, initialise training with this batch as the first batch
- **is_last** (*Optional[bool]*) – Set to `True` if current batch is the last in training. This has the same effect as if data from both trainings were presented at once.
- **train_biases** (*Optional[bool]*) – If `True`, train biases as if they were weights. Otherwise present biases will be ignored in training and not be changed. Default: `True`, train biases as well as weights
- **calc_intermediate_results** (*Optional[bool]*) – If `True`, calculates the intermediate weights not in the final batch. Default: `False`, do not compute intermediate weights
- **return_training_progress** (*Optional[bool]*) – If `True`, return dict of current training variables for each batch. Default: `True`, return training progress
- **return_trained_output** (*Optional[bool]*) – If `True`, return the result of evolving the layer with the trained weights in the output dict. Default: `False`, do not return the trained output
- **fisher_relabelling** (*Optional[bool]*) – If `True`, relabel target data such that the training algorithm is equivalent to Fisher discriminant analysis. Default: `False`, use standard ridge / linear regression
- **standardize** (*Optional[bool]*) – Train with z-score standardized data, based on means and standard deviations from first batch. Default: `False`, do not standardize data

Returns If `return_training_progress` is `True`, return a dict with current training variables (xtx, xty, kahan_comp_xtx, kahan_comp_xty). Weights and biases are returned if `is_last` is `True` or if `calc_intermediate_results` is `True`. If `return_trained_output` is `True`, the dict contains the output of evolving the layer with the newly trained weights.

property weights

(ndarray) Weights encapsulated by this layer (MxN)

12.4.4 API reference for layers.FFIAFBrian

class layers.**FFIAFBrian**(*weights: numpy.ndarray, bias: Union[float, numpy.ndarray, None] = 15. * mamp, dt: Optional[float] = 100. * usecond, noise_std: Optional[float] = 0. * volt, tau_mem: Union[float, numpy.ndarray, None] = 20. * msecond, v_thresh: Union[float, numpy.ndarray, None] = -55. * mvolt, v_reset: Union[float, numpy.ndarray, None] = -65. * mvolt, v_rest: Union[float, numpy.ndarray, None] = -65. * mvolt, refractory: Optional[float] = 0. * second, neuron_eq: Optional[str] = <Equations object consisting of 8 equations>, integrator_name: Optional[str] = 'rk4', name: Optional[str] = 'unnamed', record: Optional[bool] = False*)

Bases: rockpool.layers.layer.Layer

DEPRECATED A spiking feedforward layer with current inputs and spiking outputs

__init__(*weights: numpy.ndarray, bias: Union[float, numpy.ndarray, None] = 15. * mamp, dt: Optional[float] = 100. * usecond, noise_std: Optional[float] = 0. * volt, tau_mem: Union[float, numpy.ndarray, None] = 20. * msecond, v_thresh: Union[float, numpy.ndarray, None] = -55. * mvolt, v_reset: Union[float, numpy.ndarray, None] = -65. * mvolt, v_rest: Union[float, numpy.ndarray, None] = -65. * mvolt, refractory: Optional[float] = 0. * second, neuron_eq: Optional[str] = <Equations object consisting of 8 equations>, integrator_name: Optional[str] = 'rk4', name: Optional[str] = 'unnamed', record: Optional[bool] = False*)

Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** (*np.array*) – Layer weight matrix [N_in, N]
- **bias** (*Optional[np.array]*) – Nx1 bias vector. Default: `10mA`
- **dt** (*Optional[float]*) – Time-step. Default: `0.1 ms`
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default:‘‘ 0.‘‘
- **tau_mem** (*Optional[FloatVector]*) – Nx1 vector of neuron time constants. Default: `20ms`
- **v_thresh** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-55mV`
- **v_reset** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-65mV`
- **v_rest** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-65mV`
- **refractory** (*Optional[float]*) – Refractory period after each spike. Default: `0ms`
- **str] neuron_eq** (*Optional[Brian2.Equations,]*) – Set of neuron equations. Default: IAF equation set
- **integrator_name** (*Optional[str]*) – Integrator to use for simulation. Default: `'rk4'`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`
- **record** (*Optional[bool]*) – Record membrane potential during evolutions

Attributes

bias	
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output time series class for this layer (*TSEvent*)
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	
v_reset	
v_rest	
v_thresh	
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of the layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
stream(duration, dt[, verbose])	Stream data through this layer
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights: numpy.ndarray*, *bias: Union[float*, *numpy.ndarray*, *None] = 15. * mamp*, *dt: Optional[float] = 100. * usecond*, *noise_std: Optional[float] = 0. * volt*, *tau_mem: Union[float*, *numpy.ndarray*, *None] = 20. * msecond*, *v_thresh: Union[float*, *numpy.ndarray*, *None] = -55. * mvolt*, *v_reset: Union[float*, *numpy.ndarray*, *None] = -65. * mvolt*, *v_rest: Union[float*, *numpy.ndarray*, *None] = -65. * mvolt*, *refractory: Optional[float] = 0. * second*, *neuron_eq: Optional[str] = <Equations object consisting of 8 equations>*, *integrator_name: Optional[str] = 'rk4'*, *name: Optional[str] = 'unnamed'*, *record: Optional[bool] = False*)

Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** (*np.array*) – Layer weight matrix [N_in, N]
- **bias** (*Optional[np.array]*) – Nx1 bias vector. Default: `10mA`
- **dt** (*Optional[float]*) – Time-step. Default: `0.1 ms`
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default:‘‘ 0.‘‘
- **tau_mem** (*Optional[FloatVector]*) – Nx1 vector of neuron time constants. Default: `20ms`
- **v_thresh** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-55mV`
- **v_reset** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-65mV`
- **v_rest** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `-65mV`
- **refractory** (*Optional[float]*) – Refractory period after each spike. Default: `0ms`
- **str] neuron_eq** (*Optional[Brian2.Equations,*) – Set of neuron equations. Default: IAF equation set
- **integrator_name** (*Optional[str]*) – Integrator to use for simulation. Default: `'rk4'`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`
- **record** (*Optional[bool]*) – Record membrane potential during evolutions

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **Optional[bool]verbose** – Currently no effect, just for conformity

Return TSEvent Output spike series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass

- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output time series class for this layer (*TSEvent*)

randomize_state()
Randomize the internal state of the layer

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

stream(*duration: float*, *dt: float*, *verbose: bool = False*) → Tuple[float, List[float]]
Stream data through this layer

Parameters

- **duration** (*float*) – Total duration for which to handle streaming
- **dt** (*float*) – Streaming time step
- **verbose** (*bool*) – Display feedback

Yield (event_times, event_channels)

Returns Final (event_times, event_channels)

property t
(float) The current evolution time of this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.5 API reference for layers.FFIAFSpkInBrian

class layers.**FFIAFSpkInBrian**(*weights: numpy.ndarray*, *bias: numpy.ndarray = 10. * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn: numpy.ndarray = 20. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 10 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)
Bases: `rockpool.layers.gpl.iaf_brian.FFIAFBrian`

DEPRECATED Spiking feedforward layer with spiking inputs and outputs

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 10. * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn: numpy.ndarray = 20. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 10 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)
Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end. In- and outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms

- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set
- **integrator_name** – str Integrator to use for simulation. Default: 'rk4'
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Attributes

bias	
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output time series class for this layer (*TSEvent*)
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	
tau_syn	
v_reset	
v_rest	
v_thresh	
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end.
evolve([ts_input, duration, num_timesteps, ...])	evolve : Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Continued on next page

Table 28 – continued from previous page

load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
pot_kernel(t)	pot_kernel - response of the membrane potential to an incoming spike at a single synapse with weight 1*amp (not considering v_rest)
randomize_state()	.randomize_state() - arguments:: randomize the internal state of the layer Usage: .randomize_state()
reset_all([keep_params])	Reset both the internal clock and the internal state of the layer
reset_state()	.reset_state() - arguments:: reset the internal state of the layer Usage: .reset_state()
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
stream(duration, dt[, verbose])	Stream data through this layer
to_dict()	to_dict - Convert parameters of `self` to a dict if they are relevant for
train(ts_target, ts_input, is_first, is_last)	train - Wrapper to standardize training syntax across layers. Use
train_mst_simple(duration, t_start, ts_input)	train_mst_simple - Use the multi-spike tempotron learning rule

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 10. * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn: numpy.ndarray = 20. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 10 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Construct a spiking feedforward layer with IAF neurons, with a Brian2 back-end. In- and outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set
- **integrator_name** – str Integrator to use for simulation. Default: 'rk4'
- **name** – str Name for the layer. Default: 'unnamed'

- **record** – bool Record membrane potential during evolutions

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If True, then None is permitted as argument for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of .dt. If not provided, then duration or the duration of ts_input will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of .dt

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a TSEvent time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of .dt. If not provided, then either duration or the duration of ts_input will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of .dt

property class_name

(str) Class name of self

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

evolve : Function to evolve the states of this layer given an input

Parameters

- **tsSpkInput** – TSEvent Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output time series class for this layer (*TSEvent*)

pot_kernel(*t*)
pot_kernel - response of the membrane potential to an incoming spike at a single synapse with weight 1*amp (not considering v_rest)

randomize_state()
.randomize_state() - arguments:: randomize the internal state of the layer Usage: .randomize_state()

reset_all(*keep_params=True*)
Reset both the internal clock and the internal state of the layer

reset_state()
.reset_state() - arguments:: reset the internal state of the layer Usage: .reset_state()

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

stream (*duration: float*, *dt: float*, *verbose: bool = False*) → Tuple[float, List[float]]
Stream data through this layer

Parameters

- **duration** (*float*) – Total duration for which to handle streaming
- **dt** (*float*) – Streaming time step
- **verbose** (*bool*) – Display feedback

Yield (event_times, event_channels)

Returns Final (event_times, event_channels)

property t
(float) The current evolution time of this layer

to_dict () → dict

to_dict - Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

train (*ts_target: None*, *ts_input: rockpool.timeseries.TSContinuous*, *is_first: bool*, *is_last: bool*, *method: str = 'mst'*, ***kwargs*)

train - Wrapper to standardize training syntax across layers. Use specified training method to train layer for current batch.

Parameters

- **ts_target** – Target time series for current batch. Can be skipped for `mst` method.
- **ts_input** – Input to the layer during the current batch.
- **is_first** – Set `True` to indicate that this batch is the first in training procedure.
- **is_last** – Set `True` to indicate that this batch is the last in training procedure.
- **method** – String indicating which training method to choose. Currently only multi-spike tempotron ("mst") is supported.

kwargs will be passed on to corresponding training method. For 'mst' method, kwargs `duration` and `t_start` must be provided.

train_mst_simple(*duration: float*, *t_start: float*, *ts_input: rockpool.timeseries.TSEvent*, *target_counts: numpy.ndarray = None*, *lambda_: float = 1e-05*, *eligibility_ratio: float = 0.1*, *momentum: float = 0*, *is_first: bool = True*, *is_last: bool = False*, *verbose: bool = False*)

train_mst_simple - Use the multi-spike tempotron learning rule from Guetig2017, in its simplified version, where no gradients are calculated

property weights

(ndarray) Weights encapsulated by this layer (MxN)

12.4.6 API reference for layers.RecIAFBrian

class layers.**RecIAFBrian**(*weights: numpy.ndarray = None*, *bias: Union[float, numpy.ndarray] = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: Union[float, numpy.ndarray] = 20. * msecond*, *tau_syn_r: Union[float, numpy.ndarray] = 50. * msecond*, *v_thresh: Union[float, numpy.ndarray] = -55. * mvolt*, *v_reset: Union[float, numpy.ndarray] = -65. * mvolt*, *v_rest: Union[float, numpy.ndarray] = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 8 equations>*, *rec_syn_eq: str = <Equations object consisting of 2 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Bases: `rockpool.layers.layer.Layer`

DEPRECATED A spiking recurrent layer with current inputs and spiking outputs, using a Brian2 backend

__init__(*weights: numpy.ndarray = None*, *bias: Union[float, numpy.ndarray] = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: Union[float, numpy.ndarray] = 20. * msecond*, *tau_syn_r: Union[float, numpy.ndarray] = 50. * msecond*, *v_thresh: Union[float, numpy.ndarray] = -55. * mvolt*, *v_reset: Union[float, numpy.ndarray] = -65. * mvolt*, *v_rest: Union[float, numpy.ndarray] = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 8 equations>*, *rec_syn_eq: str = <Equations object consisting of 2 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end. Current input, spiking output

Parameters

- **weights** – np.array NxN weight matrix. Default: [100x100] unit-lambda matrix
- **bias** – np.array Nx1 bias vector. Default: 10.5mA
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20 ms
- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 50 ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set

- **rec_syn_eq** – Brian2.Equations set of synapse equations for recurrent connects. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Attributes

bias	(np.ndarray) Bias currents for the neurons in this layer [N,]
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output time series data type for this layer (*TSEvent*)
refractory	(np.ndarray) Refractory period for the neurons in this layer [N,]
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Membrane potential for the neurons in this layer [N,]
t	(float) Current layer time in s
tau_mem	(np.ndarray) Membrane time constants for the neurons in this layer [N,]
tau_syn_r	(np.ndarray) Synaptic time constants for recurrent synapses in this layer [N**2,]
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_rest	(np.ndarray) Resting potential for the neurons in this layer [N,]
v_thresh	(np.ndarray) Threshold potentials for the neurons in this layer [N,]
weights	(np.ndarray) Recurrent weights for this layer

Methods

__init__([weights, bias, dt, noise_std, ...])	Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input

Continued on next page

Table 30 – continued from previous page

load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of the layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights: numpy.ndarray = None*, *bias: Union[float*, *numpy.ndarray] = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: Union[float*, *numpy.ndarray] = 20. * msecond*, *tau_syn_r: Union[float*, *numpy.ndarray] = 50. * msecond*, *v_thresh: Union[float*, *numpy.ndarray] = -55. * mvolt*, *v_reset: Union[float*, *numpy.ndarray] = -65. * mvolt*, *v_rest: Union[float*, *numpy.ndarray] = -65. * mvolt*, *refractory: float = 0. * second*, *neuron_eq: str = <Equations object consisting of 8 equations>*, *rec_syn_eq: str = <Equations object consisting of 2 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end. Current input, spiking output

Parameters

- **weights** – np.array NxN weight matrix. Default: [100x100] unit-lambda matrix
- **bias** – np.array Nx1 bias vector. Default: 10.5mA
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20 ms
- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 50 ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set
- **rec_syn_eq** – Brian2.Equations set of synapse equations for recurrent connects. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of .dt. If not provided, then duration or the duration of ts_input will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of .dt

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a TSEvent time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of .dt. If not provided, then either duration or the duration of ts_input will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of .dt

property bias

(np.ndarray) Bias currents for the neurons in this layer [N,]

property class_name

(str) Class name of self

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*TSContinuous*) – Input currents
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return TSEvent: output spike series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output time series data type for this layer (*TSEvent*)

randomize_state()
Randomize the internal state of the layer

property refractory
(np.ndarray) Refractory period for the neurons in this layer [N,]

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Membrane potential for the neurons in this layer [N,]

property t
(float) Current layer time in s

property tau_mem
(np.ndarray) Membrane time constants for the neurons in this layer [N,]

property tau_syn_r
(np.ndarray) Synaptic time constants for recurrent synapses in this layer [N**2,]

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_rest
(np.ndarray) Resting potential for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potentials for the neurons in this layer [N,]

property weights
(np.ndarray) Recurrent weights for this layer

12.4.7 API reference for layers.RecIAFSpkInBrian

class layers.**RecIAFSpkInBrian**(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: numpy.ndarray = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn_inp: numpy.ndarray = 50. * msecond*, *tau_syn_rec: numpy.ndarray = 50. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory=0. * second*, *neuron_eq=<Equations object consisting of 9 equations>*, *synapse_eq=<Equations object consisting of 4 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Bases: `rockpool.layers.gpl.iaf_brian.RecIAFBrian`

DEPRECATED Spiking recurrent layer with spiking in- and outputs, and a Brian2 backend

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: numpy.ndarray = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn_inp: numpy.ndarray = 50. * msecond*, *tau_syn_rec: numpy.ndarray = 50. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory=0. * second*, *neuron_eq=<Equations object consisting of 9 equations>*, *synapse_eq=<Equations object consisting of 4 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end. In- and outputs are spiking events

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10.5mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set
- **synapse_eq** – Brian2.Equations set of synapse equations for recurrent connects. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'rk4'
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Attributes

bias	(np.ndarray) Bias currents for the neurons in this layer [N,]
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(~.TSEvent') Input time series class accepted by this layer (*TSEvent*)
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output time series data type for this layer (*TSEvent*)

Continued on next page

Table 31 – continued from previous page

refractory	(np.ndarray) Refractory period for the neurons in this layer [N,]
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Membrane potential for the neurons in this layer [N,]
t	(float) Current layer time in s
tau_mem	(np.ndarray) Membrane time constants for the neurons in this layer [N,]
tau_syn_inp	(np.ndarray) Input synaptic time constants for this layer [M, N]
tau_syn_r	(np.ndarray) Synaptic time constants for recurrent synapses in this layer [N**2,]
tau_syn_rec	(np.ndarray) Recurrent synaptic time constants for this layer [N, N]
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_rest	(np.ndarray) Resting potential for the neurons in this layer [N,]
v_thresh	(np.ndarray) Threshold potentials for the neurons in this layer [N,]
weights	(np.ndarray) Recurrent synaptic weights for this layer [N, N]
weights_in	(np.ndarray) Input weights for this layer [M, N]
weights_rec	(np.ndarray) Recurrent synaptic weights for this layer [N, N]

Methods

__init__(weights_in, weights_rec[, bias, ...])	Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of the layer
reset_all([keep_params])	Reset all state of this layer (time and internal state)
reset_state()	Reset the internal state of the layer
reset_time()	Reset the time for this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file

Continued on next page

Table 32 – continued from previous page

to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: numpy.ndarray = 10.5 * mamp*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_mem: numpy.ndarray = 20. * msecond*, *tau_syn_inp: numpy.ndarray = 50. * msecond*, *tau_syn_rec: numpy.ndarray = 50. * msecond*, *v_thresh: numpy.ndarray = -55. * mvolt*, *v_reset: numpy.ndarray = -65. * mvolt*, *v_rest: numpy.ndarray = -65. * mvolt*, *refractory=0. * second*, *neuron_eq=<Equations object consisting of 9 equations>*, *synapse_eq=<Equations object consisting of 4 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*, *record: bool = False*)

Construct a spiking recurrent layer with IAF neurons, with a Brian2 back-end. In- and outputs are spiking events

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10.5mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **neuron_eq** – Brian2.Equations set of neuron equations. Default: IAF equation set
- **synapse_eq** – Brian2.Equations set of synapse equations for recurrent connects. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'rk4'
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time

- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(np.ndarray) Bias currents for the neurons in this layer [N,]

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*TSEvent*) – Input spike train
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return TSEvent: output spike series

property input_type

(~.TSEvent‘) Input time series class accepted by this layer (*TSEvent*)

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`

- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output time series data type for this layer (*TSEvent*)

randomize_state()
Randomize the internal state of the layer

property refractory
(np.ndarray) Refractory period for the neurons in this layer [N,]

reset_all(*keep_params=True*)
Reset all state of this layer (time and internal state)

reset_state()
Reset the internal state of the layer

reset_time()
Reset the time for this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Membrane potential for the neurons in this layer [N,]

property t
(float) Current layer time in s

property tau_mem
(np.ndarray) Membrane time constants for the neurons in this layer [N,]

property tau_syn_inp
(np.ndarray) Input synaptic time constants for this layer [M, N]

property tau_syn_r
(np.ndarray) Synaptic time constants for recurrent synapses in this layer [N**2,]

property tau_syn_rec
(np.ndarray) Recurrent synaptic time constants for this layer [N, N]

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_rest
(np.ndarray) Resting potential for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potentials for the neurons in this layer [N,]

property weights
(np.ndarray) Recurrent synaptic weights for this layer [N, N]

property weights_in
(np.ndarray) Input weights for this layer [M, N]

property weights_rec
(np.ndarray) Recurrent synaptic weights for this layer [N, N]

12.4.8 API reference for layers.PassThroughEvents

class layers.**PassThroughEvents**(*weights: numpy.ndarray*, *dt: float = 0.001*, *noise_std: Optional[float] = None*, *name: str = 'unnamed'*)
Bases: `rockpool.layers.layer.Layer`

Pass through events by routing to different channels

__init__(*weights: numpy.ndarray*, *dt: float = 0.001*, *noise_std: Optional[float] = None*, *name: str = 'unnamed'*)
Pass through events by routing to different channels

Parameters

- **weights** – np.ndarray Positive integer weight matrix for this layer
- **dt** – float Time step duration. Only used for determining evolution period and internal clock.

- **noise_std** – float Not actually used
- **name** – str Name of this layer. Default: 'unnamed'

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, dt, noise_std, name])	Pass through events by routing to different channels
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *dt: float = 0.001*, *noise_std: Optional[float] = None*, *name: str = 'unnamed'*)

Pass through events by routing to different channels

Parameters

- **weights** – np.ndarray Positive integer weight matrix for this layer

- **dt** – float Time step duration. Only used for determining evolution period and internal clock.
- **noise_std** – float Not actually used
- **name** – str Name of this layer. Default: 'unnamed'

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds

- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **tsSpkInput** – TSEvent Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.9 API reference for layers.FFExpSynBrian

class layers.**FFExpSynBrian**(*weights: Union[numpy.ndarray, int] = None*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_syn: float = 5. * msecond*, *synapse_eq=<Equations object consisting of 2 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*)
Bases: `rockpool.layers.layer.Layer`

DEPRECATED Define an exponential synapse layer (spiking input), with a Brian2 backend

__init__(*weights: Union[numpy.ndarray, int] = None*, *dt: float = 100. * usecond*, *noise_std: float = 0. * volt*, *tau_syn: float = 5. * msecond*, *synapse_eq=<Equations object consisting of 2 equations>*, *integrator_name: str = 'rk4'*, *name: str = 'unnamed'*)
Construct an exponential synapse layer (spiking input), with a Brian2 backend

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer
- **dt** – float Time step for state evolution. Default: 0.1 ms
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0

- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_syn	
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__([weights, dt, noise_std, tau_syn, ...])	Construct an exponential synapse layer (spiking input), with a Brian2 backend
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of the layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: Union[numpy.ndarray, int] = None, dt: float = 100. * usecond, noise_std: float = 0. * volt, tau_syn: float = 5. * msecond, synapse_eq=<Equations object consisting of 2 equations>, integrator_name: str = 'rk4', name: str = 'unnamed'*)
Construct an exponential synapse layer (spiking input), with a Brian2 backend

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer
- **dt** – float Time step for state evolution. Default: 0.1 ms
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0
- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp, var_name: str = 'input', allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace (*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)
Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous
Function to evolve the states of this layer given an input

Parameters

- **ts_input** – TSEvent Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return TSContinuous: output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of the layer

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

abstract to_dict() → dict
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.10 API reference for layers.FFExpSyn

class layers.**FFExpSyn**(*weights: Union[numpy.ndarray, int]*, *bias: Union[numpy.ndarray, float] = 0*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_syn: float = 0.005*, *name: str = 'unnamed'*, *add_events: bool = True*)
Bases: `rockpool.layers.training.gpl.rr_trained_layer.RRTrainedLayer`

Define an exponential synapse layer with spiking inputs and current outputs

__init__(*weights: Union[numpy.ndarray, int], bias: Union[numpy.ndarray, float] = 0, dt: float = 0.0001, noise_std: float = 0, tau_syn: float = 0.005, name: str = 'unnamed', add_events: bool = True*)

Construct an exponential synapse layer (spiking inputs, current outputs)

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer
- **dt** – float Time step for state evolution
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0
- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one.

Attributes

bias	(np.ndarray) Bias currents for the neurons in this layer [N,]
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Time series class accepted by this layer (*TSEvent*)
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal neuron state of the neurons in this layer [N,]
t	(float) The current evolution time of this layer
tau_syn	(float) Output synaptic time constants for this layer
weights	(ndarray) Weights encapsulated by this layer (MxN)
xtx	(np.ndarray) $X^{T}X$ intermediate training value
xty	(np.ndarray) $X^{T}Y$ intermediate training value

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct an exponential synapse layer (spiking inputs, current outputs)
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer
train(ts_target, ts_input, is_first, is_last)	Wrapper to standardize training syntax across layers.
train_logreg(ts_target[, ts_input, ...])	Train self with logistic regression over one of possibly many batches.
train_rr(ts_target[, ts_input, regularize, ...])	Train self with ridge regression over one of possibly many batches.

__init__(*weights: Union[numpy.ndarray, int], bias: Union[numpy.ndarray, float] = 0, dt: float = 0.0001, noise_std: float = 0, tau_syn: float = 0.005, name: str = 'unnamed', add_events: bool = True*)

Construct an exponential synapse layer (spiking inputs, current outputs)

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer
- **dt** – float Time step for state evolution
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0
- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one.

_batch_update(*inp: numpy.ndarray*, *target: numpy.ndarray*, *reset: bool*, *train_biases: bool*, *standardize: bool*, *update_weights: bool*, *return_training_progress: bool*) → Dict

Train with the already processed input and target data of the current batch. Update layer weights and biases if requested. Provide information on training state if requested.

Parameters

- **inp** (*np.ndarray*) – 2D-array (num_samples x num_features) of input data.

- **target** (*np.ndarray*) – 2D-array (num_samples x 'self.size') of target data.
- **reset** (*bool*) – If 'True', internal variables will be reset at the end.
- **train_bises** (*bool*) – Should biases be trained or only weights?
- **standardize** (*bool*) – Has input data been z-score standardized?
- **update_weights** (*bool*) – Set 'True' to update layer weights and biases.
- **return_training_progress** (*bool*) – Return intermediate training data (e.g. xtx, xty,...)

Return dict Dict with information on training progress, depending on values of other function arguments.

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp, var_name: str = 'input', allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size

- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_filter_data(*data: numpy.ndarray*, *num_timesteps: int*)
Filter input data y convolving with the synaptic kernel

Parameters

- **data** (*np.ndarray*) – Input data
- **num_timesteps** (*int*) – The number of time steps to return

Return np.ndarray The filtered data

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_gradients(*inp: numpy.ndarray*, *target: numpy.ndarray*, *regularize: float*) → numpy.ndarray
Compute gradients for this batch

Parameters

- **inp** (*np.ndarray*) – Input time series for this batch [T, M]
- **target** (*np.ndarray*) – Target time series for this batch [T, O]
- **regularize** (*float*) – Regularization parameter for weights. Reduces the L1-norm of the weights (weight sum)

Return np.ndarray Gradients for weights

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input and return as raster

Parameters

- **ts_input** (*Optional[TSEvent]*) – Spiking input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Raster containing spike info
num_timesteps: (np.ndarray) Number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_training_data(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*)

Check and rasterize input and target signals for this batch

Parameters

- **ts_target** (`TSContinuous`) – Target signal for this batch
- **ts_input** (*Optional[TSEvent]*) – Input signal for this batch. Default: `None`, no input for this batch
- **is_first** (*Optional[bool]*) – If `True`, this is the first batch in training. Default: `True`, this is the first batch
- **is_last** (*optional[bool]*) – If `True`, this is the last training batch. Default: `False`, this is not the last batch

:return (inp, target, time_base) inp np.ndarray: Rasterized input signal [T, M] target np.ndarray: Rasterized target signal [T, O] time_base np.ndarray: Time base for `inp` and `target`

property bias

(np.ndarray) Bias currents for the neurons in this layer [N,]

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input spike train
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSContinuous Output currents

property input_type

(*TSEvent*) Time series class accepted by this layer (*TSEvent*)

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal neuron state of the neurons in this layer [N,]

property t
(float) The current evolution time of this layer

property tau_syn
(float) Output synaptic time constants for this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer

train(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSContinuous*, *is_first: bool*, *is_last: bool*, *method: str = 'rr'*, ***kwargs*)
Wrapper to standardize training syntax across layers. Use specified training method to train layer for current batch.

Parameters

- **ts_target** (TSContinuous) – Target time series for current batch.
- **ts_input** (TSContinuous) – Input to the layer during the current batch.
- **is_first** (*bool*) – Set `True` to indicate that this batch is the first in training procedure.
- **is_last** (*bool*) – Set `True` to indicate that this batch is the last in training procedure.
- **method** (*str*) – String indicating which training method to choose. Currently only ridge regression ("rr") and logistic regression are supported.
- **kwargs** – Will be passed on to corresponding training method.

train_logreg(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSEvent = None*, *learning_rate: float = 0*, *regularize: float = 0*, *batch_size: Optional[int] = None*, *epochs: int = 1*, *store_states: bool = True*, *verbose: bool = False*)
Train self with logistic regression over one of possibly many batches. Note that this training method assumes that a sigmoid function is applied to the layer output, which is not the case in *evolve*.

Parameters

- **ts_target** (TSContinuous) – Target for current batch
- **ts_input** (TSEvent) – Input to self for current batch
- **learning_rate** (*float*) – Factor determining scale of weight increments at each step
- **regularize** (*float*) – Regularization parameter
- **batch_size** (*int*) – Number of samples per batch. If None, train with all samples at once
- **epochs** (*int*) – How many times is training repeated

- **store_states** (*bool*) – Include last state from previous training and store state from this training. This has the same effect as if data from both trainings were presented at once.
- **verbose** (*bool*) – Print output about training progress

train_rr (*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Union[rockpool.timeseries.TSEvent, rockpool.timeseries.TSContinuous] = None*, *regularize: float = 0*, *is_first: bool = True*, *is_last: bool = False*, *store_states: bool = True*, *train_biases: bool = True*, *calc_intermediate_results: bool = False*, *return_training_progress: bool = True*, *return_trained_output: bool = False*, *fisher_relabelling: bool = False*, *standardize: bool = False*) → Optional[Dict]

Train self with ridge regression over one of possibly many batches. Use Kahan summation to reduce rounding errors when adding data to existing matrices from previous batches.

Parameters

- **ts_target** (TSContinuous) – Target for current batch
- **TSContinuous] ts_input** (*Union[TSEvent,*) – Input to self for current batch
- **regularize** (*float*) – Regularization parameter for ridge regression
- **is_first** (*bool*) – `True` if current batch is the first in training
- **is_last** (*bool*) – `True` if current batch is the last in training
- **store_states** (*bool*) – If `True`, include last state from previous training and store state from this training. This has the same effect as if data from both trainings were presented at once.
- **train_biases** (*bool*) – If `True`, train biases as if they were weights Otherwise present biases will be ignored in training and not be changed.
- **calc_intermediate_results** (*bool*) – If `True`, calculates the intermediate weights not in the final batch
- **return_training_progress** (*bool*) – If `True`, return dict of current training variables for each batch.
- **standardize** (*bool*) – If `True`, train with z-score standardized data, based on means and standard deviations from first batch

Return Union[None, dict] If `return_training_progress`, return dict with current training variables (xtx, xty, kahan_comp_xtx, kahan_comp_xty). Weights and biases are returned if `is_last` or if `calc_intermediate_results`. If `return_trained_output`, the dict contains the output of evolving with the newly trained weights.

property weights

(ndarray) Weights encapsulated by this layer (MxN)

property xtx

(np.ndarray) $X^{T}X$ intermediate training value

property xty

(np.ndarray) $X^{T}Y$ intermediate training value

12.4.11 API reference for layers.RecLIFJax

class layers.**RecLIFJax**(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

Bases: `rockpool.layers.layer.Layer`

Recurrent spiking neuron layer (LIF), spiking input and spiking output. No input / output weights.

RecLIFJax is a basic recurrent spiking neuron layer, implemented with a JAX-backed Euler solver backend. Outputs are spikes generated by each layer neuron; no output weighting is provided. Inputs are provided by spiking through a synapse onto each layer neuron; no input weighting is provided. The layer is therefore N inputs -> N neurons -> N outputs.

This layer can be used to implement gradient-based learning systems, using the JAX-provided automatic differentiation functionality of `jax.grad`.

Dynamics

The dynamics of the N neurons' membrane potential V_{mem} and the N synaptic currents I_{syn} evolve under the system

$$\begin{aligned} \tau_{syn}\dot{I}_{syn} + I_{syn} &= 0 \\ I_{syn} &+= S_{in}(t) \\ \tau_{syn}\dot{V}_{mem} + V_{mem} &= I_{syn} + b + \sigma\zeta(t) \end{aligned}$$

where $S_{in}(t)$ is a vector containing `1` for each input channel that emits a spike at time t; b is a N vector of bias currents for each neuron; $\sigma\zeta(t)$ is a white-noise process with standard deviation σ injected independently onto each neuron's membrane; and τ_{mem} and τ_{syn} are the membrane and synaptic time constants, respectively.

On spiking

When the membane potential for neuron j, $V_{mem,j}$ exceeds the threshold voltage $V_{thr} = 0$, then the neuron emits a spike.

$$\begin{aligned} V_{mem,j} > V_{thr} &\rightarrow S_{rec,j} = 1 \\ I_{syn} &= I_{syn} + S_{rec} \cdot w_{rec} \\ V_{mem,j} &= V_{mem,j} - 1 \end{aligned}$$

Neurons therefore share a common resting potential of `0`, a firing threshold of `0`, and a subtractive reset of `-1`. Neurons each have an optional bias current *bias* (default: `-1`).

Surrogate signals

To facilitate gradient-based training, a surrogate $U(t)$ is generated from the membrane potentials of each neuron.

$$U_j = \text{sig}(V_j)$$

Where $\text{sig}(x) = (1 + \exp(-x))^{-1}$.

Outputs from evolution

As output, this layer returns the spiking activity of the N neurons from the *evolve* method. After each evolution, the attributes `spikes_last_evolution`, `i_rec_last_evolution` and `v_mem_last_evolution` and `surrogate_last_evolution` will be *TimeSeries* objects containing the appropriate time series.

__init__(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

Attributes

bias	(ndarray) Bias current for each neuron [N,]
class_name	(str) Class name of `self`
dt	(float) Forward Euler solver time step
input_type	*TSEvent*
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	*TSEvent*
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	(ndarray) Membrane time constant for each neuron [N,]
tau_syn	(ndarray) Output synaptic time constant for each neuron [N,]
w_recurrent	(ndarray) Recurrent weight matrix [N, N]
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

`__init__`(w_recurrent, tau_mem, tau_syn[, ...])	A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.
`evolve`([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
`load_from_dict`(config, **kwargs)	Generate instance of a `Layer` subclass with parameters loaded from a dictionary
`load_from_file`(filename, **kwargs)	Generate an instance of a `Layer` subclass, with parameters loaded from a file
`randomize_state`()	Randomize the internal state of this layer
`reset_all`()	Reset both the internal clock and the internal state of the layer
`reset_state`()	Reset the membrane potentials, synaptic currents and refractory state for this layer
`reset_time`()	Reset the internal clock of this layer to 0
`save`(config, filename)	Save a set of parameters to a `json` file
`save_layer`(filename)	Obtain layer paramters from `to_dict` and save in a `json` file
`to_dict`()	Convert the configuration of this layer into a dictionary to assist in reconstruction

__init__(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*sp_input_ts: jax.numpy.lax_numpy.ndarray*, *I_input_ts: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]

Raw evolution over an input array

Parameters

- **sp_input_ts** (*ndarray*) – Input matrix [T, I]
- **I_input_ts** (*ndarray*) – Input matrix [T, N]

Returns (Irec_ts, output_ts, surrogate_ts, spike_raster_ts, Vmem_ts, Isyn_ts) Irec_ts: (np.ndarray) Time trace of recurrent current inputs per neuron [T, N] output_ts: (np.ndarray) Time trace of surrogate weighted output [T, O] surrogate_ts: (np.ndarray) Time trace of surrogate from each neuron [T, N] spike_raster_ts: (np.ndarray) Boolean raster [T, N]; `True` if a spike occurred in time step `t`, from neuron `n` Vmem_ts: (np.ndarray) Time trace of neuron membrane potentials [T, N] Isyn_ts: (np.ndarray) Time trace of output synaptic currents [T, N]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds

- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)*

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>)*

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(ndarray) Bias current for each neuron [N,]

property class_name

(str) Class name of `self`

property dt

(float) Forward Euler solver time step

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSEvent

Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input time series. Default: `None`, no stimulus is provided
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time
- **Optional[bool]verbose** – Currently no effect, just for conformity

Return TSContinuous Output time series; the synaptic currents of each neuron

property input_type

TSEvent

Type (TSEvent) Input *TimeSeries* class

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls

Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std

(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type

TSEvent

Type (TSEvent) Output *TimeSeries* class

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the membrane potentials, synaptic currents and refractory state for this layer

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

property tau_mem
(ndarray) Membrane time constant for each neuron [N,]

property tau_syn
(ndarray) Output synaptic time constant for each neuron [N,]

to_dict() → dict
Convert the configuration of this layer into a dictionary to assist in reconstruction

Returns dict

property w_recurrent
(ndarray) Recurrent weight matrix [N, N]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.12 API reference for layers.RecLIFCurrentInJax

class layers.**RecLIFCurrentInJax**(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

Bases: `rockpool.layers.gpl.lif_jax.RecLIFJax`

Recurrent spiking neuron layer (LIF), current injection input and spiking output. No input / output weights.

RecLIFCurrentInJax is a basic recurrent spiking neuron layer, implemented with a JAX-backed Euler solver backend. Outputs are spikes generated by each layer neuron; no output weighting is provided. Inputs are provided by direct current injection onto each neuron membrane; no input weighting is provided. The layer is therefore N inputs -> N neurons -> N outputs.

This layer can be used to implement gradient-based learning systems, using the JAX-provided automatic differentiation functionality of `jax.grad`.

Dynamics

The dynamics of the N neurons' membrane potential V_{mem} and the N synaptic currents I_{syn} evolve under the system

$$\tau_{syn}\dot{I}_{syn} + I_{syn} = 0$$
$$\tau_{syn}\dot{V}_{mem} + V_{mem} = I_{syn} + I_{in}(t) + b + \sigma\zeta(t)$$

where $S_{in}(t)$ is a vector containing `1` for each input channel that emits a spike at time t; $I_{in}(t)$ is a vector of input currents injected directly onto the neuron membranes; b is a N vector of bias currents for each neuron; $\sigma\zeta(t)$ is a white-noise process with standard deviation σ injected independently onto each neuron's membrane; and τ_{mem} and τ_{syn} are the membrane and synaptic time constants, respectively.

On spiking

When the membane potential for neuron j, $V_{mem,j}$ exceeds the threshold voltage $V_{thr} = 0$, then the neuron emits a spike.

$$V_{mem,j} > V_{thr} \rightarrow S_{rec,j} = 1$$
$$I_{syn} = I_{syn} + S_{rec} \cdot w_{rec}$$
$$V_{mem,j} = V_{mem,j} - 1$$

Neurons threfore share a common resting potential of `0`, a firing threshold of `0`, and a subtractive reset of `-1`. Neurons each have an optional bias current *bias* (default: `-1`).

Surrogate signals

To facilitate gradient-based training, a surrogate $U(t)$ is generated from the membrane potentials of each neuron.

$$U_j = \text{sig}(V_j)$$

Where $\text{sig}(x) = (1 + \exp(-x))^{-1}$.

Outputs from evolution

As output, this layer returns the spiking activity of the N neurons from the *evolve* method. After each evolution, the attributes `spikes_last_evolution`, `i_rec_last_evolution` and `v_mem_last_evolution` and `surrogate_last_evolution` will be *TimeSeries* objects containing the appropriate time series.

__init__(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

Attributes

bias	(ndarray) Bias current for each neuron [N,]
class_name	(str) Class name of `self`
dt	(float) Forward Euler solver time step
input_type	*TSContinuous*
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	*TSEvent*
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	(ndarray) Membrane time constant for each neuron [N,]
tau_syn	(ndarray) Output synaptic time constant for each neuron [N,]
w_recurrent	(ndarray) Recurrent weight matrix [N, N]
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

`__init__`(w_recurrent, tau_mem, tau_syn[, ...])	A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.
`evolve`([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
`load_from_dict`(config, **kwargs)	Generate instance of a `Layer` subclass with parameters loaded from a dictionary
`load_from_file`(filename, **kwargs)	Generate an instance of a `Layer` subclass, with parameters loaded from a file
`randomize_state`()	Randomize the internal state of this layer
`reset_all`()	Reset both the internal clock and the internal state of the layer
`reset_state`()	Reset the membrane potentials, synaptic currents and refractory state for this layer
`reset_time`()	Reset the internal clock of this layer to 0
`save`(config, filename)	Save a set of parameters to a `json` file
`save_layer`(filename)	Obtain layer paramters from `to_dict` and save in a `json` file
`to_dict`()	Convert the configuration of this layer into a dictionary to assist in reconstruction

__init__(*w_recurrent: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*sp_input_ts: jax.numpy.lax_numpy.ndarray*, *I_input_ts: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]

Raw evolution over an input array

Parameters

- **sp_input_ts** (*ndarray*) – Input matrix [T, I]
- **I_input_ts** (*ndarray*) – Input matrix [T, N]

Returns (Irec_ts, output_ts, surrogate_ts, spike_raster_ts, Vmem_ts, Isyn_ts) Irec_ts: (np.ndarray) Time trace of recurrent current inputs per neuron [T, N] output_ts: (np.ndarray) Time trace of surrogate weighted output [T, O] surrogate_ts: (np.ndarray) Time trace of surrogate from each neuron [T, N] spike_raster_ts: (np.ndarray) Boolean raster [T, N]; `True` if a spike occurred in time step `t`, from neuron `n` Vmem_ts: (np.ndarray) Time trace of neuron membrane potentials [T, N] Isyn_ts: (np.ndarray) Time trace of output synaptic currents [T, N]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds

- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)*

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>)*

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(ndarray) Bias current for each neuron [N,]

property class_name

(str) Class name of `self`

property dt

(float) Forward Euler solver time step

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSEvent

Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series. Default: `None`, no stimulus is provided
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSEvent Output time series; spiking activity each neuron

property input_type

TSContinuous

Type (TSContinuous) Output *TimeSeries* class

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls

Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std

(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type

TSEvent

Type (TSEvent) Output *TimeSeries* class

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the membrane potentials, synaptic currents and refractory state for this layer

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

property tau_mem
(ndarray) Membrane time constant for each neuron [N,]

property tau_syn
(ndarray) Output synaptic time constant for each neuron [N,]

to_dict() → dict
Convert the configuration of this layer into a dictionary to assist in reconstruction

Returns dict

property w_recurrent
(ndarray) Recurrent weight matrix [N, N]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.13 API reference for layers.RecLIFJax_IO

class layers.**RecLIFJax_IO**(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

Bases: `rockpool.layers.gpl.lif_jax.RecLIFJax`

Recurrent spiking neuron layer (LIF), spiking input and weighted surrogate output. Input and output weights.

RecLIFJax is a basic recurrent spiking neuron layer, implemented with a JAX-backed Euler solver backend. Outputs are surrogates generated by each layer neuron, weighted by a set of ourput weights. Inputs are provided by spiking through a synapse onto each layer neuronvia a set of input weights. The layer is therefore M inputs -> N neurons -> O outputs.

This layer can be used to implement gradient-based learning systems, using the JAX-provided automatic differentiation functionality of `jax.grad`.

Dynamics

The dynamics of the N neurons' membrane potential V_{mem} and the N synaptic currents I_{syn} evolve under the system

$$\begin{aligned}\tau_{syn}\dot{I}_{syn} + I_{syn} &= 0\\ I_{syn} &+= S_{in}(t)\cdot w_{in}\\ \tau_{syn}\dot{V}_{mem} + V_{mem} &= I_{syn} + I_{in}(t)\cdot w_{in} + b + \sigma\zeta(t)\end{aligned}$$

where $S_{in}(t)$ is a vector containing `1` for each input channel that emits a spike at time t; w_{in} is a $[N_{in} \times N]$ matrix of input weights; $I_{in}(t)$ is a vector of input currents injected directly onto the neuron membranes; b is a N vector of bias currents for each neuron; $\sigma\zeta(t)$ is a white-noise process with standard deviation σ injected independently onto each neuron's membrane; and τ_{mem} and τ_{syn} are the membrane and synaptic time constants, respectively.

On spiking

When the membane potential for neuron j, $V_{mem,j}$ exceeds the threshold voltage $V_{thr} = 0$, then the neuron emits a spike.

$$\begin{aligned}V_{mem,j} > V_{thr} &\rightarrow S_{rec,j} = 1\\ I_{syn} &= I_{syn} + S_{rec}\cdot w_{rec}\\ V_{mem,j} &= V_{mem,j} - 1\end{aligned}$$

Neurons threfore share a common resting potential of `0`, a firing threshold of `0`, and a subtractive reset of `-1`. Neurons each have an optional bias current *bias* (default: `-1`).

Surrogate signals

To facilitate gradient-based training, a surrogate $U(t)$ is generated from the membrane potentials of each neuron. This is used to provide a weighted output $O(t)$.

$$\begin{aligned}U_j &= \text{sig}(V_j)\\ O(t) &= U(t)\cdot w_{out}\end{aligned}$$

Where w_{out} is a $[N \times N_{out}]$ matrix of output weights, and $\text{sig}(x) = (1 + \exp(-x))^{-1}$.

Outputs from evolution

As output, this layer returns the weighted surrogate activity of the N neurons from the *evolve* method. After each evolution, the attributes `spikes_last_evolution`, `i_rec_last_evolution` and `v_mem_last_evolution` and `surrogate_last_evolution` will be *TimeSeries* objects containing the appropriate time series.

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

Attributes

bias	(ndarray) Bias current for each neuron [N,]
class_name	(str) Class name of `self`
dt	(float) Forward Euler solver time step
input_type	*TSEvent*
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	*TSContinuous*
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	(ndarray) Membrane time constant for each neuron [N,]
tau_syn	(ndarray) Output synaptic time constant for each neuron [N,]
w_in	(np.ndarray) [M,N] input weights

Continued on next page

Table 43 – continued from previous page

w_out	(np.ndarray) [N,O] output weights
w_recurrent	(ndarray) Recurrent weight matrix [N, N]
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(w_in, w_recurrent, w_out, tau_mem, ...)	A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.
evolve([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the membrane potentials, synaptic currents and refractory state for this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the configuration of this layer into a dictionary to assist in reconstruction

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*sp_input_ts: jax.numpy.lax_numpy.ndarray*, *I_input_ts: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]

Raw evolution over an input array

Parameters

- **sp_input_ts** (*ndarray*) – Input matrix [T, I]
- **I_input_ts** (*ndarray*) – Input matrix [T, N]

Returns (Irec_ts, output_ts, surrogate_ts, spike_raster_ts, Vmem_ts, Isyn_ts) Irec_ts: (np.ndarray) Time trace of recurrent current inputs per neuron [T, N] output_ts: (np.ndarray) Time trace of surrogate weighted output [T, O] surrogate_ts: (np.ndarray) Time trace of surrogate from each neuron [T, N] spike_raster_ts: (np.ndarray) Boolean raster [T, N]; True if a spike occurred in time step t, from neuron n Vmem_ts: (np.ndarray) Time trace of neuron membrane potentials [T, N] Isyn_ts: (np.ndarray) Time trace of output synaptic currents [T, N]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size

- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)
Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias
(ndarray) Bias current for each neuron [N,]

property class_name
(str) Class name of `self`

property dt
(float) Forward Euler solver time step

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSContinuous
Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input time series. Default: `None`, no stimulus is provided
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time
- **Optional[bool]verbose** – Currently no effect, just for conformity

Return TSContinuous Output time series; the synaptic currents of each neuron

property input_type
TSEvent

Type (TSEvent) Input *TimeSeries* class

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
TSContinuous

Type (TSContinuous) Output *TimeSeries* class

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the membrane potentials, synaptic currents and refractory state for this layer

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

property tau_mem
(ndarray) Membrane time constant for each neuron [N,]

property tau_syn
(ndarray) Output synaptic time constant for each neuron [N,]

to_dict() → dict
Convert the configuration of this layer into a dictionary to assist in reconstruction

Returns dict

property w_in
(np.ndarray) [M,N] input weights

property w_out
(np.ndarray) [N,O] output weights

property w_recurrent
(ndarray) Recurrent weight matrix [N, N]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.14 API reference for layers.RecLIFCurrentInJax_IO

class layers.**RecLIFCurrentInJax_IO**(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)
Bases: rockpool.layers.gpl.lif_jax.RecLIFJax_IO

Recurrent spiking neuron layer (LIF), weighted current input and weighted surrogate output. Input / output weighting provided.

RecLIFJax is a basic recurrent spiking neuron layer, implemented with a JAX-backed Euler solver backend. Outputs are surrogates generated by each layer neuron, via a set of output weights. Inputs are provided by weighted current injection to each layer neuron, via a set of input weights. The layer is therefore M inputs -> N neurons -> O outputs.

This layer can be used to implement gradient-based learning systems, using the JAX-provided automatic differentiation functionality of jax.grad.

Dynamics

The dynamics of the N neurons' membrane potential V_{mem} and the N synaptic currents I_{syn} evolve under the system

$$\tau_{syn}\dot{I}_{syn} + I_{syn} = 0$$
$$I_{syn} += S_{in}(t) \cdot w_{in}$$
$$\tau_{syn}\dot{V}_{mem} + V_{mem} = I_{syn} + I_{in}(t) \cdot w_{in} + b + \sigma\zeta(t)$$

where $S_{in}(t)$ is a vector containing 1 for each input channel that emits a spike at time t; w_{in} is a $[N_{in} \times N]$ matrix of input weights; $I_{in}(t)$ is a vector of input currents injected directly onto the neuron membranes; b is a N vector of bias currents for each neuron; $\sigma\zeta(t)$ is a white-noise process with standard deviation σ injected independently onto each neuron's membrane; and τ_{mem} and τ_{syn} are the membrane and synaptic time constants, respectively.

On spiking

When the membane potential for neuron j, $V_{mem,j}$ exceeds the threshold voltage $V_{thr} = 0$, then the neuron emits a spike.

$$V_{mem,j} > V_{thr} \rightarrow S_{rec,j} = 1$$
$$I_{syn} = I_{syn} + S_{rec} \cdot w_{rec}$$
$$V_{mem,j} = V_{mem,j} - 1$$

Neurons therefore share a common resting potential of `0`, a firing threshold of `0`, and a subtractive reset of `-1`. Neurons each have an optional bias current *`bias`* (default: `-1`).

Surrogate signals

To facilitate gradient-based training, a surrogate $U(t)$ is generated from the membrane potentials of each neuron. This is used to provide a weighted output $O(t)$.

$$U_j = \text{sig}(V_j)$$
$$O(t) = U(t) \cdot w_{out}$$

Where w_{out} is a $[N \times N_{out}]$ matrix of output weights, and $\text{sig}(x) = (1 + \exp(-x))^{-1}$.

Outputs from evolution

As output, this layer returns the weighted surrogate activity of the N neurons from the *`evolve`* method. After each evolution, the attributes `spikes_last_evolution`, `i_rec_last_evolution` and `v_mem_last_evolution` and `surrogate_last_evolution` will be *`TimeSeries`* objects containing the appropriate time series.

`__init__`(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **`w_recurrent`** (*`ndarray`*) – [N,N] Recurrent weight matrix
- **`tau_mem`** (*`ArrayLike[float]`*) – [N,] Membrane time constants
- **`tau_syn`** (*`ArrayLike[float]`*) – [N,] Output synaptic time constants
- **`bias`** (*`Optional[ArrayLike[float]]`*) – [N,] Bias currents for each neuron (Default: 0)
- **`noise_std`** (*`Optional[float]`*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **`dt`** (*`Optional[float]`*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **`name`** (*`Optional[str]`*) – Name of this layer. Default: `None`
- **`rng_key`** (*`Optional[int]`*) – JAX pRNG key. Default: generate a new key

Attributes

`bias`	(ndarray) Bias current for each neuron [N,]
`class_name`	(str) Class name of `self`

Continued on next page

Table 45 – continued from previous page

dt	(float) Forward Euler solver time step
input_type	*TSContinuous*
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	*TSContinuous*
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_mem	(ndarray) Membrane time constant for each neuron [N,]
tau_syn	(ndarray) Output synaptic time constant for each neuron [N,]
w_in	(np.ndarray) [M,N] input weights
w_out	(np.ndarray) [N,O] output weights
w_recurrent	(ndarray) Recurrent weight matrix [N, N]
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(w_in, w_recurrent, w_out, tau_mem, ...)	A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.
evolve([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the membrane potentials, synaptic currents and refractory state for this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the configuration of this layer into a dictionary to assist in reconstruction

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau_mem: Union[float, jax.numpy.lax_numpy.ndarray], tau_syn: Union[float, jax.numpy.lax_numpy.ndarray], bias: Union[float, jax.numpy.lax_numpy.ndarray, None] = -1.0, noise_std: Optional[float] = 0.0, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

A basic recurrent spiking neuron layer, with a JAX-implemented forward Euler solver.

Parameters

- **w_recurrent** (*ndarray*) – [N,N] Recurrent weight matrix
- **tau_mem** (*ArrayLike[float]*) – [N,] Membrane time constants
- **tau_syn** (*ArrayLike[float]*) – [N,] Output synaptic time constants
- **bias** (*Optional[ArrayLike[float]]*) – [N,] Bias currents for each neuron (Default: 0)
- **noise_std** (*Optional[float]*) – Std. dev. of white noise injected independently onto the membrane of each neuron (Default: 0)
- **dt** (*Optional[float]*) – Forward Euler solver time step. Default: min(tau_mem, tau_syn) / 10
- **name** (*Optional[str]*) – Name of this layer. Default: `None`
- **rng_key** (*Optional[int]*) – JAX pRNG key. Default: generate a new key

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*sp_input_ts: jax.numpy.lax_numpy.ndarray*, *I_input_ts: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]
Raw evolution over an input array

Parameters

- **sp_input_ts** (*ndarray*) – Input matrix [T, I]
- **I_input_ts** (*ndarray*) – Input matrix [T, N]

Returns (Irec_ts, output_ts, surrogate_ts, spike_raster_ts, Vmem_ts, Isyn_ts) Irec_ts: (np.ndarray) Time trace of recurrent current inputs per neuron [T, N] output_ts: (np.ndarray) Time trace of surrogate weighted output [T, O] surrogate_ts: (np.ndarray) Time trace of surrogate from each neuron [T, N] spike_raster_ts: (np.ndarray) Boolean raster [T, N]; `True` if a spike occurred in time step `t`, from neuron `n` Vmem_ts: (np.ndarray) Time trace of neuron

membrane potentials [T, N] Isyn_ts: (np.ndarray) Time trace of output synaptic currents [T, N]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias
(ndarray) Bias current for each neuron [N,]

property class_name
(str) Class name of `self`

property dt
(float) Forward Euler solver time step

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSEvent
Evolve the state of this layer given an input

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input time series. Default: `None`, no stimulus is provided
- **duration** (*Optional[float]*) – Simulation/Evolution time, in seconds. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of *dt*. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSEvent Output time series; spiking activity each neuron

property input_type
TSContinuous

Type (TSContinuous) Output *TimeSeries* class

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
TSContinuous

Type (TSContinuous) Output *TimeSeries* class

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the membrane potentials, synaptic currents and refractory state for this layer

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

property tau_mem
(ndarray) Membrane time constant for each neuron [N,]

property tau_syn
(ndarray) Output synaptic time constant for each neuron [N,]

to_dict() → dict
Convert the configuration of this layer into a dictionary to assist in reconstruction

Returns dict

property w_in
(np.ndarray) [M,N] input weights

property w_out
(np.ndarray) [N,O] output weights

property w_recurrent
(ndarray) Recurrent weight matrix [N, N]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.15 API reference for layers.FFCLIAF

class layers.**FFCLIAF**(*weights: numpy.ndarray*, *bias: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_thresh: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *v_reset: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_subtract: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *dt: Optional[float] = 1.0*, *monitor_id: Union[int*, *None*, *numpy.ndarray*, *List*, *Tuple] = []*, *name: Optional[str] = 'unnamed'*)
Bases: `rockpool.layers.gpl.iaf_cl.CLIAF`

Feedforward layer of integrate and fire neurons with constant leak

__init__(*weights: numpy.ndarray*, *bias: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_thresh: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *v_reset: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_subtract: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *dt: Optional[float] = 1.0*, *monitor_id: Union[int*, *None*, *numpy.ndarray*, *List*, *Tuple] = []*, *name: Optional[str] = 'unnamed'*)
Feedforward layer of integrate and fire neurons with constant leak

Parameters

- **weights** (*np.ndarray*) – Input weight matrix [N_in, N]
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step [N,]. Default: `0.`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold [N,]. Default: `8.`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike [N,]. Default: `0.`
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise will reset. Default: `8.`
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`, do not record neuron state
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

Attributes

bias	(np.ndarray) Bias values for the neurons in this layer [N,]
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).
monitor_id	(list) List of neurons that should be monitored during evolution
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal state of this layer
t	(float) The current evolution time of this layer
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_subtract	(np.ndarray) Subtractive reset values for the neurons in this layer [N,]
v_thresh	(np.ndarray) Threshold potental for the neurons in this layer [N,]
weights	(ndarray) Weights encapsulated by this layer (MxN)
weights_in	(np.ndarray) Input weights for this layer [N_in, N]

Methods

`__init__`(weights[, bias, v_thresh, v_reset, ...])	Feedforward layer of integrate and fire neurons with constant leak
`evolve`([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
`load_from_dict`(config, **kwargs)	Generate instance of a `Layer` subclass with parameters loaded from a dictionary
`load_from_file`(filename, **kwargs)	Generate an instance of a `Layer` subclass, with parameters loaded from a file
`randomize_state`()	Randomize the internal state of this layer
`reset_all`()	Reset both the internal clock and the internal state of the layer
`reset_state`()	Reset the internal state of this layer
`reset_time`()	Reset the internal clock of this layer
`save`(config, filename)	Save a set of parameters to a `json` file
`save_layer`(filename)	Obtain layer paramters from `to_dict` and save in a `json` file
`to_dict`()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights: numpy.ndarray*, *bias: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_thresh: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *v_reset: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 0.0*, *v_subtract: Union[numpy.ndarray*, *List*, *Tuple*, *float*, *None] = 8.0*, *dt: Optional[float] = 1.0*, *monitor_id: Union[int*, *None*, *numpy.ndarray*, *List*, *Tuple] = []*, *name: Optional[str] = 'unnamed'*)

Feedforward layer of integrate and fire neurons with constant leak

Parameters

- **weights** (*np.ndarray*) – Input weight matrix [N_in, N]
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step [N,]. Default: `0.`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold [N,]. Default: `8.`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike [N,]. Default: `0.`
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise will reset. Default: `8.`
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`, do not record neuron state
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

_add_to_record(*state_time_series: list*, *t_now: float*, *id_out: Union[numpy.ndarray*, *List*, *Tuple*, *bool] = True*, *state: Optional[numpy.ndarray] = None*, *debug: bool = False*)

Convenience function to record current state of the layer or individual neuron

Parameters

- **state_time_series** (*list*) – A simple python list object to which the state needs to be appended
- **t_now** (*float*) – Current simulation time
- **id_out** (*Optional[np.ndarray]*) – Neuron IDs to record the state of. If `True` all the neuron's states will be added to the record. Default: `True`, record all neurons

- **state** (*Optional[np.ndarray]*) – If not `None`, record this as state, otherwise record `self.state`
- **debug** (*Optional[bool]*) – If `True`, print debug info. Default: `False`, do not print debug info

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input, set up time base

Parameters

- **ts_input** (*Optional[TSEvent]*) – TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Boolean raster containing spike info num_timesteps: (int) Number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(np.ndarray) Bias values for the neurons in this layer [N,]

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input spike train
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSEvent Output spike series

property input_type
(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property monitor_id
(list) List of neurons that should be monitored during evolution

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved

- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of this layer

property t
(float) The current evolution time of this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_subtract
(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potental for the neurons in this layer [N,]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

property weights_in
(np.ndarray) Input weights for this layer [N_in, N]

12.4.16 API reference for layers.RecCLIAF

class layers.**RecCLIAF**(*weights_in: numpy.ndarray, weights_rec: numpy.ndarray, bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, refractory: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, dt: Optional[float] = 0.0001, delay: Optional[float] = None, tTauBias: Optional[float] = None, monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = [], state_type: Union[type, str, None] = <class 'float'>, name: Optional[str] = 'unnamed'*)
Bases: `rockpool.layers.gpl.iaf_cl.CLIAF`

Recurrent layer of integrate and fire neurons with constant leak

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *refractory: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *dt: Optional[float] = 0.0001*, *delay: Optional[float] = None*, *tTauBias: Optional[float] = None*, *monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = []*, *state_type: Union[type, str, None] = <class 'float'>*, *name: Optional[str] = 'unnamed'*)

Recurrent layer of integrate and fire neurons with constant leak

Parameters

- **weights_in** (*np.ndarray*) – Input weight matrix [N_in, N]
- **weights_rec** (*np.ndarray*) – Recurrent weight matrix [N, N]
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step [N,]. Default: `0.`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold [N,]. Default: `8.`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike (also see param `v_subtract`) [N,]. Default: `0.`
- **v_subtract** (*Optional[FloatVector]*) – [N,] If not `None`, subtract provided values from neuron state after spike. Otherwise will reset. Default: `8.`
- **refractory** (*Optional[FloatVector]*) – Vector of refractory times [N,]
- **dt** (*Optional[float]*) – Time step size in s. Default: `0.1 ms`
- **delay** (*Optional[float]*) – Time after which a spike within the layer arrives at the recurrent synapses of the receiving neurons within the network. Rounded down to multiple of *dt*. Must be at least *dt*. Default: `None`, use *dt*
- **tTauBias** (*Optional[float]*) – Period for applying bias. Must be at least *dt*. Is rounded down to multiple of *dt*. If `None`, will be set to *dt*. Default: `None`, use *dt*
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`, do not monitor neurons
- **state_type** (*Optional[type]*) – Data type for the membrane potential. Default: `float`
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

Attributes

bias	(np.ndarray) Bias values for the neurons in this layer [N,]
class_name	(str) Class name of `self`
delay	(float) Event transmission delay for the events in this layer, in s
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).
monitor_id	(list) List of neurons that should be monitored during evolution

Continued on next page

Table 49 – continued from previous page

noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).
refractory	(float) Refractory period for the neurons in this layer
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal state of the neurons in this layer [N,]
state_type	(type) Data type of the neuron state in this layer
t	(float) The current evolution time of this layer
tTauBias	(float) Time step on which to apply biases
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_subtract	(np.ndarray) Subtractive reset values for the neurons in this layer [N,]
v_thresh	(np.ndarray) Threshold potental for the neurons in this layer [N,]
weights	(np.ndarray) Recurrent weights for this layer
weights_in	(np.ndarray) Input weights for this layer [N_in, N]
weights_rec	(np.ndarray) Recurrent weights for this layer

Methods

__init__(weights_in, weights_rec[, bias, ...])	Recurrent layer of integrate and fire neurons with constant leak
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *refractory: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *dt: Optional[float] = 0.0001*, *delay: Optional[float] = None*, *tTauBias: Optional[float] = None*, *monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = []*, *state_type: Union[type, str, None] = <class 'float'>*, *name: Optional[str] = 'unnamed'*)

Recurrent layer of integrate and fire neurons with constant leak

Parameters

- **weights_in** (*np.ndarray*) – Input weight matrix [N_in, N]
- **weights_rec** (*np.ndarray*) – Recurrent weight matrix [N, N]
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step [N,]. Default: `0.`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold [N,]. Default: `8.`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike (also see param `v_subtract`) [N,]. Default: `0.`
- **v_subtract** (*Optional[FloatVector]*) – [N,] If not `None`, subtract provided values from neuron state after spike. Otherwise will reset. Default: `8.`
- **refractory** (*Optional[FloatVector]*) – Vector of refractory times [N,]
- **dt** (*Optional[float]*) – Time step size in s. Default: `0.1 ms`
- **delay** (*Optional[float]*) – Time after which a spike within the layer arrives at the recurrent synapses of the receiving neurons within the network. Rounded down to multiple of *dt*. Must be at least *dt*. Default: `None`, use *dt*
- **tTauBias** (*Optional[float]*) – Period for applying bias. Must be at least *dt*. Is rounded down to multiple of *dt*. If `None`, will be set to *dt*. Default: `None`, use *dt*
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`, do not monitor neurons
- **state_type** (*Optional[type]*) – Data type for the membrane potential. Default: `float`
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

_add_to_record(*state_time_series: list*, *t_now: float*, *id_out: Union[numpy.ndarray, List, Tuple, bool] = True*, *state: Optional[numpy.ndarray] = None*, *debug: bool = False*)

Convenience function to record current state of the layer or individual neuron

Parameters

- **state_time_series** (*list*) – A simple python list object to which the state needs to be appended
- **t_now** (*float*) – Current simulation time
- **id_out** (*Optional[np.ndarray]*) – Neuron IDs to record the state of. If `True` all the neuron's states will be added to the record. Default: `True`, record all neurons
- **state** (*Optional[np.ndarray]*) – If not `None`, record this as state, otherwise record `self.state`
- **debug** (*Optional[bool]*) – If `True`, print debug info. Default: `False`, do not print debug info

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If True, then None is permitted as argument for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray inp, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (*Optional[TSEvent]*) – TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Boolean raster containing spike info num_timesteps: (int) Number of evolution time steps

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(np.ndarray) Bias values for the neurons in this layer [N,]

property class_name

(str) Class name of `self`

property delay

(float) Event transmission delay for the events in this layer, in s

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input spike trian
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Show progress bar during evolution

Return TSEvent Output spike series

property input_type

(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property monitor_id
(list) List of neurons that should be monitored during evolution

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).

randomize_state()
Randomize the internal state of this layer

property refractory
(float) Refractory period for the neurons in this layer

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of the neurons in this layer [N,]

property state_type
(type) Data type of the neuron state in this layer

property t
(float) The current evolution time of this layer

property tTauBias
(float) Time step on which to apply biases

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_subtract
(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potental for the neurons in this layer [N,]

property weights
(np.ndarray) Recurrent weights for this layer

property weights_in
(np.ndarray) Input weights for this layer [N_in, N]

property weights_rec
(np.ndarray) Recurrent weights for this layer

12.4.17 API reference for layers.CLIAF

class layers.**CLIAF**(*weights_in: numpy.ndarray, bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, dt: Optional[float] = 1.0, monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = [], name: Optional[str] = 'unnamed'*)
Bases: `rockpool.layers.layer.Layer`

Abstract layer class of integrate and fire neurons with constant leak

__init__(*weights_in: numpy.ndarray, bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0, dt: Optional[float] = 1.0, monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = [], name: Optional[str] = 'unnamed'*)

Feedforward layer of integrate and fire neurons with constant leak

Parameters

- **weights_in** (*FloatVector*) – Input weight matrix
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step. Default: `0.0`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold. Default: `8.0`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike (also see param `v_subtract`). Default: `8.0`
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise neurons will reset on each spike
- **monitor_id** (*Optional[ArrayLike[int]]*) – IDs of neurons to be recorded. Default: `[]`, do not monitor any neurons
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

Attributes

bias	(np.ndarray) Bias values for the neurons in this layer [N,]
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).
monitor_id	(list) List of neurons that should be monitored during evolution
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal state of this layer
t	(float) The current evolution time of this layer
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_subtract	(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

Continued on next page

Table 51 – continued from previous page

v_thresh	(np.ndarray) Threshold potental for the neurons in this layer [N,]
weights	(ndarray) Weights encapsulated by this layer (MxN)
weights_in	(np.ndarray) Input weights for this layer [N_in, N]

Methods

__init__(weights_in[, bias, v_thresh, ...])	Feedforward layer of integrate and fire neurons with constant leak
evolve([ts_input, duration, num_timesteps])	Abstract method to evolve the state of this layer
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights_in: numpy.ndarray*, *bias: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = 8.0*, *dt: Optional[float] = 1.0*, *monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = []*, *name: Optional[str] = 'unnamed'*)

Feedforward layer of integrate and fire neurons with constant leak

Parameters

- **weights_in** (*FloatVector*) – Input weight matrix
- **bias** (*Optional[FloatVector]*) – Constant bias to be added to state at each time step. Default: `0.0`
- **v_thresh** (*Optional[FloatVector]*) – Spiking threshold. Default: `8.0`
- **v_reset** (*Optional[FloatVector]*) – Reset potential after spike (also see param `v_subtract`). Default: `8.0`
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise neurons will reset on each spike
- **monitor_id** (*Optional[ArrayLike[int]]*) – IDs of neurons to be recorded. Default: `[]`, do not monitor any neurons
- **name** (*Optional[str]*) – Name of this layer. Default: `'unnamed'`

_add_to_record(*state_time_series: list*, *t_now: float*, *id_out: Union[numpy.ndarray, List, Tuple, bool] = True*, *state: Optional[numpy.ndarray] = None*, *debug: bool = False*)

Convenience function to record current state of the layer or individual neuron

Parameters

- **state_time_series** (*list*) – A simple python list object to which the state needs to be appended
- **t_now** (*float*) – Current simulation time
- **id_out** (*Optional[np.ndarray]*) – Neuron IDs to record the state of. If `True` all the neuron's states will be added to the record. Default: `True`, record all neurons
- **state** (*Optional[np.ndarray]*) – If not `None`, record this as state, otherwise record `self.state`
- **debug** (*Optional[bool]*) – If `True`, print debug info. Default: `False`, do not print debug info

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input, set up time base

Parameters

- **ts_input** (*Optional[TSEvent]*) – TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Boolean raster containing spike info num_timesteps: (int) Number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias
(np.ndarray) Bias values for the neurons in this layer [N,]

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

abstract evolve(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → rockpool.timeseries.TimeSeries
Abstract method to evolve the state of this layer

This method must be overridden to produce a concrete *Layer* subclass. The *evolve* method is the main interface for simulating a layer. It must accept an input time series which determines the signals injected into the layer as input, and return an output time series representing the output of the layer.

Parameters

- **ts_input** (*Optional[TimeSeries]*) – (TxM) External input trace to use when evolving the layer
- **duration** (*Optional[float]*) – Duration in seconds to evolve the layer. If not provided, then `num_timesteps` or the duration of `ts_input` is used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of time steps to evolve the layer, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` is used to determine evolution time

Return TimeSeries (TxN) Output of this layer

property input_type
(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property monitor_id
(list) List of neurons that should be monitored during evolution

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output subclass emitted by this layer (*TSEvent*).

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of this layer

property t
(float) The current evolution time of this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

Return dict Dictionary of layer parameters

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_subtract
(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potental for the neurons in this layer [N,]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

property weights_in
(np.ndarray) Input weights for this layer [N_in, N]

12.4.18 API reference for layers.SoftMaxLayer

class layers.**SoftMaxLayer**(*weights: Optional[numpy.ndarray] = None*, *thresh: Optional[float] = 10000000000.0*, *dt: Optional[float] = 1.0*, *name: Optional[str] = 'unnamed'*)
Bases: rockpool.layers.gpl.iaf_cl.FFCLIAF

A spiking SoftMax layer with spiking inputs and outputs, and constant leak

This layer implements an approximation of the "soft-max" function used often in deep classification networks.

__init__(*weights: Optional[numpy.ndarray] = None*, *thresh: Optional[float] = 10000000000.0*, *dt: Optional[float] = 1.0*, *name: Optional[str] = 'unnamed'*)
Implements a spiking softmax on the inputs

Parameters

- **weights** (*Optional[np.ndarray]*) – Weight matrix. Default: None
- **thresh** (*Optional[float]*) – Spiking threshold. Default: 1e10
- **dt** (*Optional[float]*) – Time step. Default: 1.
- **name** (*Optional[str]*) – Name of this layer. Default: "unnamed"

Attributes

bias	(np.ndarray) Bias values for the neurons in this layer [N,]
class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).
monitor_id	(list) List of neurons that should be monitored during evolution
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output *TimeSeries* class of this layer (*TSEvent*)
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer name attribute
state	(np.ndarray) Internal state of this layer
t	(float) The current evolution time of this layer
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_subtract	(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

Continued on next page

Table 53 – continued from previous page

v_thresh	(np.ndarray) Threshold potental for the neurons in this layer [N,]
weights	(ndarray) Weights encapsulated by this layer (MxN)
weights_in	(np.ndarray) Input weights for this layer [N_in, N]

Methods

__init__([weights, thresh, dt, name])	Implements a spiking softmax on the inputs
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights: Optional[numpy.ndarray] = None*, *thresh: Optional[float] = 10000000000.0*, *dt: Optional[float] = 1.0*, *name: Optional[str] = 'unnamed'*)

Implements a spiking softmax on the inputs

Parameters

- **weights** (*Optional[np.ndarray]*) – Weight matrix. Default: `None`
- **thresh** (*Optional[float]*) – Spiking threshold. Default: `1e10`
- **dt** (*Optional[float]*) – Time step. Default: `1.`
- **name** (*Optional[str]*) – Name of this layer. Default: `"unnamed"`

_add_to_record(*state_time_series: list*, *t_now: float*, *id_out: Union[numpy.ndarray, List, Tuple, bool] = True*, *state: Optional[numpy.ndarray] = None*, *debug: bool = False*)

Convenience function to record current state of the layer or individual neuron

Parameters

- **state_time_series** (*list*) – A simple python list object to which the state needs to be appended
- **t_now** (*float*) – Current simulation time
- **id_out** (*Optional[np.ndarray]*) – Neuron IDs to record the state of. If `True` all the neuron's states will be added to the record. Default: `True`, record all neurons
- **state** (*Optional[np.ndarray]*) – If not `None`, record this as state, otherwise record `self.state`

- **debug** (*Optional[bool]*) – If `True`, print debug info. Default: `False`, do not print debug info

_check_input_dims (*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size (*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape (*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input, set up time base

Parameters

- **ts_input** (*Optional[TSEvent]*) – TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Boolean raster containing spike info num_timesteps: (int) Number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property bias

(np.ndarray) Bias values for the neurons in this layer [N,]

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input spike trian
- **duration** (*Optional[float]*) – Simulation/Evolution time

:param Optional[int] num_timesteps Number of evolution time steps :param Optional[bool] verbose: Currently no effect, just for conformity :return TSEvent: Output spike series

property input_type

(*TSEvent*) Input subclass accepted by this layer (*TSEvent*).

classmethod load_from_dict(*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property monitor_id
(list) List of neurons that should be monitored during evolution

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output *TimeSeries* class of this layer (*TSEvent*)

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

reset_time()
Reset the internal clock of this layer

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of this layer

property t
(float) The current evolution time of this layer

to_dict () → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_subtract
(np.ndarray) Subtractive reset values for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potental for the neurons in this layer [N,]

property weights
(ndarray) Weights encapsulated by this layer (MxN)

property weights_in
(np.ndarray) Input weights for this layer [N_in, N]

12.4.19 API reference for layers.RecDIAF

class layers.**RecDIAF**(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *dt: Optional[float] = 0.0001*, *delay: Optional[float] = 1e-08*, *tau_leak: Optional[float] = 0.001*, *refractory: Union[numpy.ndarray, List, Tuple, float, None] = 1e-09*, *v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 100.0*, *v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_rest: Union[numpy.ndarray, List, Tuple, float, None] = None*, *leak: Union[numpy.ndarray, List, Tuple, float, None] = 1*, *v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = None*, *state_type: Union[type, str, None] = 'int8'*, *monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = []*, *name: Optional[str] = 'unnamed'*)
Bases: `rockpool.layers.layer.Layer`

Define a spiking recurrent layer based on quantized digital IAF neurons

__init__(*weights_in: numpy.ndarray, weights_rec: numpy.ndarray, dt: Optional[float] = 0.0001, delay: Optional[float] = 1e-08, tau_leak: Optional[float] = 0.001, refractory: Union[numpy.ndarray, List, Tuple, float, None] = 1e-09, v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 100.0, v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0, v_rest: Union[numpy.ndarray, List, Tuple, float, None] = None, leak: Union[numpy.ndarray, List, Tuple, float, None] = 1, v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = None, state_type: Union[type, str, None] = 'int8', monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = [], name: Optional[str] = 'unnamed'*)

Construct a spiking recurrent layer with digital IAF neurons

Parameters

- **weights_in** (*np.array*) – nSizeInxN input weight matrix.
- **weights_rec** (*np.array*) – NxN weight matrix
- **dt** (*Optional[float]*) – Length of single time step in s. Default: `0.1 ms`
- **delay** (*Optional[float]*) – Time after which a spike within the layer arrives at the recurrent synapses of the receiving neurons within the network. Default: `1e-8`
- **tau_leak** (*Optional[float]*) – Period for applying leak in s. Default: `1 ms`
- **refractory** (*Optional[FloatVector]*) – Nx1 vector of refractory times. Default: `1 ns`
- **v_thresh** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `100.`
- **v_reset** (*Optional[FloatVector]*) – Nx1 vector of neuron reset potentials. Default: `0.`
- **v_rest** (*Optional[FloatVector]*) – Nx1 vector of neuron resting potentials. Leak will change sign for neurons with state below this. If `None`, leak will not change sign. Default: `None`
- **leak** (*Optional[FloatVector]*) – Nx1 vector of leak values. Default: `None`, no leak
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise will reset to *v_reset*.
- **state_type** (*Optional*) – Data type for the membrane potential. Default: `"int8"`
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`

Attributes

class_name	(str) Class name of `self`
delay	(float) Spiking delay for this layer
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Input time series class for this layer (‘.TSEvent)
leak	(np.ndarray) Leak for the neurons in this layer [N,]
monitor_id	(list) Neurons to monitor during evolution

Continued on next page

Table 55 – continued from previous page

noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(*TSEvent*) Output time series class for this layer (*TSEvent*)
refractory	(np.ndarray) Refractory period for the neurons in this layer [N,]
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal state of this layer [N,]
state_type	Type of neuron state for this layer (e.g.
t	(float) The current evolution time of this layer
tau_leak	(float) Leak period for this layer
v_reset	(np.ndarray) Reset potential for the neurons in this layer [N,]
v_rest	(float) Resting potential for the neurons in this layer [N,]
v_subtract	(np.ndarray) Subtractive reset for the neurons in this layer [N,]
v_thresh	(np.ndarray) Threshold potential for this layer [N,]
weights	(np.ndarray) Recurrent weights for this layer [N, N]
weights_in	(np.ndarray) Input weights for this layer [N_in,]
weights_rec	(np.ndarray) Recurrent weights for this layer [N, N]

Methods

__init__(weights_in, weights_rec[, dt, ...])	Construct a spiking recurrent layer with digital IAF neurons
evolve([ts_input, duration, num_timesteps, ...])	Evolve the state of this layer
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Set layer states to random values
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *dt: Optional[float] = 0.0001*, *delay: Optional[float] = 1e-08*, *tau_leak: Optional[float] = 0.001*, *refractory: Union[numpy.ndarray, List, Tuple, float, None] = 1e-09*, *v_thresh: Union[numpy.ndarray, List, Tuple, float, None] = 100.0*, *v_reset: Union[numpy.ndarray, List, Tuple, float, None] = 0.0*, *v_rest: Union[numpy.ndarray, List, Tuple, float, None] = None*, *leak: Union[numpy.ndarray, List, Tuple, float, None] = 1*, *v_subtract: Union[numpy.ndarray, List, Tuple, float, None] = None*, *state_type: Union[type, str, None] = 'int8'*, *monitor_id: Union[int, None, numpy.ndarray, List, Tuple] = []*, *name: Optional[str] = 'unnamed'*)

Construct a spiking recurrent layer with digital IAF neurons

Parameters

- **weights_in** (*np.array*) – nSizeInxN input weight matrix.
- **weights_rec** (*np.array*) – NxN weight matrix
- **dt** (*Optional[float]*) – Length of single time step in s. Default: `0.1 ms`
- **delay** (*Optional[float]*) – Time after which a spike within the layer arrives at the recurrent synapses of the receiving neurons within the network. Default: `1e-8`
- **tau_leak** (*Optional[float]*) – Period for applying leak in s. Default: `1 ms`
- **refractory** (*Optional[FloatVector]*) – Nx1 vector of refractory times. Default: `1 ns`
- **v_thresh** (*Optional[FloatVector]*) – Nx1 vector of neuron thresholds. Default: `100.`
- **v_reset** (*Optional[FloatVector]*) – Nx1 vector of neuron reset potentials. Default: `0.`
- **v_rest** (*Optional[FloatVector]*) – Nx1 vector of neuron resting potentials. Leak will change sign for neurons with state below this. If `None`, leak will not change sign. Default: `None`
- **leak** (*Optional[FloatVector]*) – Nx1 vector of leak values. Default: `None`, no leak
- **v_subtract** (*Optional[FloatVector]*) – If not `None`, subtract provided values from neuron state after spike. Otherwise will reset to *v_reset*.
- **state_type** (*Optional*) – Data type for the membrane potential. Default: `"int8"`
- **monitor_id** (*Optional[ArrayLike]*) – IDs of neurons to be recorded. Default: `[]`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>, <class 'float'>)

Sample input, set up time base

Parameters

- **ts_input** (*Optional[TSEvent]*) – TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (event_times, event_channels, num_timesteps, t_final) event_times: (np.ndarray) Event times event_channels: (np.ndarray) Event channels num_timesteps: (int) Number of evlution time steps t_final: (float) End time of evolution

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name

(str) Class name of `self`

property delay

(float) Spiking delay for this layer

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*) → rockpool.timeseries.TSEvent

Evolve the state of this layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – Input spike trian
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSEvent Output spike series

property input_type

(*TSEvent*) Input time series class for this layer (‘.TSEvent)

property leak

(np.ndarray) Leak for the neurons in this layer [N,]

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass

- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property monitor_id
(list) Neurons to monitor during evolution

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(*TSEvent*) Output time series class for this layer (*TSEvent*)

randomize_state()
Set layer states to random values

property refractory
(np.ndarray) Refractory period for the neurons in this layer [N,]

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of this layer [N,]

property state_type
Type of neuron state for this layer (e.g. `int8`)

property t
(float) The current evolution time of this layer

property tau_leak
(float) Leak period for this layer

to_dict() → dict
Convert parameters of `self` to a dict if they are relevant for reconstructing an identical layer.

property v_reset
(np.ndarray) Reset potential for the neurons in this layer [N,]

property v_rest
(float) Resting potential for the neurons in this layer [N,]

property v_subtract
(np.ndarray) Subtractive reset for the neurons in this layer [N,]

property v_thresh
(np.ndarray) Threshold potential for this layer [N,]

property weights
(np.ndarray) Recurrent weights for this layer [N, N]

property weights_in
(np.ndarray) Input weights for this layer [N_in,]

property weights_rec
(np.ndarray) Recurrent weights for this layer [N, N]

12.4.20 API reference for layers.RecFSSpikeEulerBT

class layers.**RecFSSpikeEulerBT**(*weights_fast: numpy.ndarray = None*, *weights_slow: numpy.ndarray = None*, *bias: numpy.ndarray = 0.0*, *noise_std: float = 0.0*, *tau_mem: Union[numpy.ndarray, float] = 0.02*, *tau_syn_r_fast: Union[numpy.ndarray, float] = 0.001*, *tau_syn_r_slow: Union[numpy.ndarray, float] = 0.1*, *v_thresh: Union[numpy.ndarray, float] = -0.055*, *v_reset: Union[numpy.ndarray, float] = -0.065*, *v_rest: Union[numpy.ndarray, float] = -0.065*, *refractory: float = -2.220446049250313e-16*, *spike_callback: Callable = None*, *dt: float = None*, *name: str = None*)
Bases: `rockpool.layers.layer.Layer`

Implement a spiking reservoir with tight E/I balance.

This class does NOT use a Brian2 back-end. See the class code for possibilities to modify neuron and synapse dynamics. Currently uses leaky IAF neurons and exponential current synapses. Note that network parameters

are tightly constrained for the reservoir to work as desired. See the documentation and source publications for details.

__init__(*weights_fast: numpy.ndarray = None*, *weights_slow: numpy.ndarray = None*, *bias: numpy.ndarray = 0.0*, *noise_std: float = 0.0*, *tau_mem: Union[numpy.ndarray, float] = 0.02*, *tau_syn_r_fast: Union[numpy.ndarray, float] = 0.001*, *tau_syn_r_slow: Union[numpy.ndarray, float] = 0.1*, *v_thresh: Union[numpy.ndarray, float] = -0.055*, *v_reset: Union[numpy.ndarray, float] = -0.065*, *v_rest: Union[numpy.ndarray, float] = -0.065*, *refractory: float = -2.220446049250313e-16*, *spike_callback: Callable = None*, *dt: float = None*, *name: str = None*)

Implement a spiking reservoir with fast and slow recurrent synapses, and a custom solver with precise spike timing.

Parameters

- **weights_fast** (*ndarray*) – [NxN] Recurrent weight matrix (fast synapses). Default: `None`
- **weights_slow** (*ndarray*) – [NxN] Recurrent weight matrix (slow synapses). Default: `None`
- **bias** (*Optional[ArrayLike[float]]*) – [Nx1] Bias currents for each neuron. Default: 0., no biases
- **noise_std** (*Optional[float]*) – Noise Std. Dev. Default: 0., no noise
- **tau_mem** (*ArrayLike[float]*) – [Nx1] Neuron time constants. Default: `20 ms`
- **tau_syn_r_fast** (*ArrayLike[float]*) – [Nx1] Post-synaptic neuron fast synapse TCs. Default: `1 ms`
- **tau_syn_r_slow** (*ArrayLike[float]*) – [Nx1] Post-synaptic neuron slow synapse TCs. Default: `100 ms`
- **v_thresh** (*ArrayLike[float]*) – [Nx1] Neuron firing thresholds. Default: `-55 mV`
- **v_reset** (*ArrayLike[float]*) – [Nx1] Neuron reset potentials. Default: `-65 mV`
- **v_rest** (*ArrayLike[float]*) – [Nx1] Neuron rest potentials. Default: `-65 mV`
- **refractory** (*Optional[float]*) – Post-spike refractory period. Default: `0.`, no refractoriness
- **spike_callback** (*Callable[]*) – Callable(lyrSpikeBT, t_time, nSpikeInd). Spike-based learning callback function. Default: `None`.
- **dt** (*Optional[float]*) – Nominal time step (Euler solver). Default: `None`, choose a reasonable *dt* as `min(tau)`
- **name** (*Optional[str]*) – Name of this layer. Default: `None`

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

Continued on next page

Table 57 – continued from previous page

noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(`TSEvent`) Output `TimeSeries` class (`TSEvent`)
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau_syn_r_f	(flot) Fast synaptic time constant (s)
tau_syn_r_s	(float) Slow synaptic time constant (s)
v_reset	(float) Reset potential
v_rest	(float) Resting potential
v_thresh	(float) Threshold potential
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__([weights_fast, weights_slow, bias, ...])	Implement a spiking reservoir with fast and slow recurrent synapses, and a custom solver with precise spike timing.
evolve([ts_input, duration, num_timesteps, ...])	Simulate the spiking reservoir, using a precise-time spike detector
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the network
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert this layer to a dictionary for saving :return dict:

__init__(*weights_fast: numpy.ndarray = None*, *weights_slow: numpy.ndarray = None*, *bias: numpy.ndarray = 0.0*, *noise_std: float = 0.0*, *tau_mem: Union[numpy.ndarray, float] = 0.02*, *tau_syn_r_fast: Union[numpy.ndarray, float] = 0.001*, *tau_syn_r_slow: Union[numpy.ndarray, float] = 0.1*, *v_thresh: Union[numpy.ndarray, float] = -0.055*, *v_reset: Union[numpy.ndarray, float] = -0.065*, *v_rest: Union[numpy.ndarray, float] = -0.065*, *refractory: float = -2.220446049250313e-16*, *spike_callback: Callable = None*, *dt: float = None*, *name: str = None*)

Implement a spiking reservoir with fast and slow recurrent synapses, and a custom solver with precise spike timing.

Parameters

- **weights_fast** (*ndarray*) – [NxN] Recurrent weight matrix (fast synapses). Default: `None`
- **weights_slow** (*ndarray*) – [NxN] Recurrent weight matrix (slow synapses). Default: `None`
- **bias** (*Optional[ArrayLike[float]]*) – [Nx1] Bias currents for each neuron. Default: 0., no biases
- **noise_std** (*Optional[float]*) – Noise Std. Dev. Default: 0., no noise
- **tau_mem** (*ArrayLike[float]*) – [Nx1] Neuron time constants. Default: `20 ms`
- **tau_syn_r_fast** (*ArrayLike[float]*) – [Nx1] Post-synaptic neuron fast synapse TCs. Default: `1 ms`
- **tau_syn_r_slow** (*ArrayLike[float]*) – [Nx1] Post-synaptic neuron slow synapse TCs. Default: `100 ms`
- **v_thresh** (*ArrayLike[float]*) – [Nx1] Neuron firing thresholds. Default: `-55 mV`
- **v_reset** (*ArrayLike[float]*) – [Nx1] Neuron reset potentials. Default: `-65 mV`
- **v_rest** (*ArrayLike[float]*) – [Nx1] Neuron rest potentials. Default: `-65 mV`
- **refractory** (*Optional[float]*) – Post-spike refractory period. Default: `0.`, no refractoriness
- **spike_callback** (*Callable[]*) – Callable(lyrSpikeBT, t_time, nSpikeInd). Spike-based learning callback function. Default: `None`.
- **dt** (*Optional[float]*) – Nominal time step (Euler solver). Default: `None`, choose a reasonable *dt* as `min(tau)`
- **name** (*Optional[str]*) – Name of this layer. Default: `None`

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

property _min_tau

(float) Smallest time constant of the layer

_prepare_input(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: Optional[bool] = False*, *min_delta: Optional[float] = None*) → rockpool.timeseries.TSEvent
Simulate the spiking reservoir, using a precise-time spike detector

This method implements an Euler integrator, coupled with precise spike time detection using a linear interpolation between integration steps. Time is then reset to the spike time, and integration proceeds. For this reason, the time steps returned by the integrator are not homogenous. A minimum time step can be set; by default this is 1/10 of the nominal time step.

Parameters

- **ts_input** (*Optional[TSContinuous]*) – Input signals over time [TxN]
- **duration** (*Optional[float]*) – Duration of simulation in seconds. Default: 100ms
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity
- **min_delta** (*Optional[float]*) – Minimum time step taken. Default: 1/10 nominal TC
- **min_delta** – Minimum time step taken. Default: 1/10 nominal TC

Return TSEvent Time series containing the output currents of the reservoir

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(`TSEvent`) Output `TimeSeries` class (`TSEvent`)

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the network

reset_time()
Reset the internal clock of this layer to 0

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

property tau_syn_r_f
(flot) Fast synaptic time constant (s)

property tau_syn_r_s
(float) Slow synaptic time constant (s)

to_dict() → dict
Convert this layer to a dictionary for saving :return dict:

property v_reset
(float) Reset potential

property v_rest
(float) Resting potential

property v_thresh
(float) Threshold potential

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.21 API reference for layers.FFUpDown

class layers.**FFUpDown**(*weights: Union[int, numpy.ndarray, Tuple[int, int]], repeat_output: Optional[int] = 1, dt: Optional[float] = 0.001, tau_decay: Union[numpy.ndarray, List, Tuple, float, None] = None, noise_std: Optional[float] = 0.0, thr_up: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, thr_down: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, name: Optional[str] = 'unnamed', max_num_timesteps: Optional[int] = 5000, multiplex_spikes: Optional[bool] = True*)
Bases: `rockpool.layers.layer.Layer`

Define a spiking feedforward layer to convert analogue inputs to up and down channels

Feedforward layer that converts each analogue input channel to one spiking up and one down channel
Runs in batch mode like `FFUpDownTorch` to save memory, but does not use pytorch. `FFUpDownTorch` seems to be slower...

__init__(*weights: Union[int, numpy.ndarray, Tuple[int, int]], repeat_output: Optional[int] = 1, dt: Optional[float] = 0.001, tau_decay: Union[numpy.ndarray, List, Tuple, float, None] = None, noise_std: Optional[float] = 0.0, thr_up: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, thr_down: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, name: Optional[str] = 'unnamed', max_num_timesteps: Optional[int] = 5000, multiplex_spikes: Optional[bool] = True*)
Construct a spiking feedforward layer to convert analogue inputs to up and down channels.

This layer is exceptional in that `state` has the same size as `size_in`, not `size`. It corresponds to the input, inferred from the output spikes by inverting the up-/down-algorithm.

Parameters

- **weights** (*np.array*) – MxN weight matrix. Unlike other *Layer* classes, the only important thing about weights its shape. The first dimension determines the number of input channels (self.size_in). The second dimension corresponds to size and has to be n*2*size_in, n up and n down channels for each input). If n>1 the up-/and down-spikes are distributed over multiple channels. The values of the weight matrix do not have any effect. It is also possible to pass only an integer, which will correspond to size_in. size is then set to 2*size_in, i.e. n=1. Alternatively a tuple of two values, corresponding to size_in and n can be passed.
- **dt** (*Optional[float]*) – Time-step. Default: 0.1 ms
- **tau_decay** (*Optional[ArrayLike]*) – The states that track the input signal for threshold comparison decay with this time constant, unless it is `None`. Default: `None`, do not decay tracking states.
- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: `0.`, no noise
- **thr_up** (*Optional[ArrayLike]*) – Thresholds for creating up-spikes. Default: `0.001`
- **thr_down** (*Optional[ArrayLike]*) – Thresholds for creating down-spikes. Default: `0.001`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`
- **max_num_timesteps** (*Optional[int]*) – Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches. Default: `MAX_NUM_TIMESTEPS_DEFAULT`
- **multiplex_spikes** (*Optional[bool]*) – If `True`, allows a channel to emit multiple spikes per time, according to how much the corresponding threshold is exceeded. Default: `True`, emit multiple spikes per time step

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)

Continued on next page

Table 59 – continued from previous page

t	(float) The current evolution time of this layer
tau_decay	
thr_down	
thr_up	
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(weights[, repeat_output, dt, ...])	Construct a spiking feedforward layer to convert analogue inputs to up and down channels.
evolve([ts_input, duration, num_timesteps, ...])	Evolve the states of this layer given an input
load_from_dict(config)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: Union[int, numpy.ndarray, Tuple[int, int]], repeat_output: Optional[int] = 1, dt: Optional[float] = 0.001, tau_decay: Union[numpy.ndarray, List, Tuple, float, None] = None, noise_std: Optional[float] = 0.0, thr_up: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, thr_down: Union[numpy.ndarray, List, Tuple, float, None] = 0.001, name: Optional[str] = 'unnamed', max_num_timesteps: Optional[int] = 5000, multiplex_spikes: Optional[bool] = True*)

Construct a spiking feedforward layer to convert analogue inputs to up and down channels.

This layer is exceptional in that `state` has the same size as `size_in`, not `size`. It corresponds to the input, inferred from the output spikes by inverting the up-/down-algorithm.

Parameters

- **weights** (*np.array*) – MxN weight matrix. Unlike other *Layer* classes, the only important thing about weights its shape. The first dimension determines the number of input channels (self.size_in). The second dimension corresponds to size and has to be n*2*size_in, n up and n down channels for each input). If n>1 the up-/and down-spikes are distributed over multiple channels. The values of the weight matrix do not have any effect. It is also possible to pass only an integer, which will correspond to size_in. size is then set to 2*size_in, i.e. n=1. Alternatively a tuple of two values, corresponding to size_in and n can be passed.
- **dt** (*Optional[float]*) – Time-step. Default: 0.1 ms
- **tau_decay** (*Optional[ArrayLike]*) – The states that track the input signal for threshold comparison decay with this time constant, unless it is `None`. Default: `None`, do not decay tracking states.

- **noise_std** (*Optional[float]*) – Noise std. dev. per second. Default: `0.`, no noise
- **thr_up** (*Optional[ArrayLike]*) – Thresholds for creating up-spikes. Default: `0.001`
- **thr_down** (*Optional[ArrayLike]*) – Thresholds for creating down-spikes. Default: `0.001`
- **name** (*Optional[str]*) – Name for the layer. Default: `'unnamed'`
- **max_num_timesteps** (*Optional[int]*) – Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches. Default: `MAX_NUM_TIMESTEPS_DEFAULT`
- **multiplex_spikes** (*Optional[bool]*) – If `True`, allows a channel to emit multiple spikes per time, according to how much the corresponding threshold is exceeded. Default: `True`, emit multiple spikes per time step

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None*) -> (*<class 'numpy.ndarray'>*, *<class 'int'>*)
_batch_data: Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace (*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'numpy.ndarray'>*, *<class 'float'>*)
Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_single_batch_evolution (*inp: numpy.ndarray*, *num_timesteps: int*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

evolve : Function to evolve the states of this layer given an input for a single batch

Parameters

- **inp** – np.ndarray Input
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Evolve the states of this layer given an input

Parameters

- **tsSpkInput** (*Optional[TSContinuous]*) – Input signal
- **duration** (*Optional[float]*) – Simulation/Evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps
- **verbose** (*Optional[bool]*) – Currently no effect, just for conformity

Return TSEvent Output spike series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

static load_from_dict (*config*)

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

static load_from_file (*filename*)

Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored

- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config*, *filename*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.22 API reference for layers.FFExpSynTorch

class layers.**FFExpSynTorch**(*weights: Union[numpy.ndarray, int] = None, bias: numpy.ndarray = 0, dt: float = 0.0001, noise_std: float = 0, tau_syn: float = 0.005, name: str = 'unnamed', add_events: bool = True, max_num_timesteps: int = 5000*)

Bases: rockpool.layers.gpl.exp_synapses_manual.FFExpSyn

Define an exponential synapse layer (spiking input, pytorch as backend)

__init__(*weights: Union[numpy.ndarray, int] = None, bias: numpy.ndarray = 0, dt: float = 0.0001, noise_std: float = 0, tau_syn: float = 0.005, name: str = 'unnamed', add_events: bool = True, max_num_timesteps: int = 5000*)

Construct an exponential synapse layer (spiking input, pytorch as backend)

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer
- **dt** – float Time step for state evolution
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0
- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one.

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

bias	
class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Time series class accepted by this layer (*TSEvent*)
max_num_timesteps	(int) Maximum number of timesteps used by the synaptic kernel
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

Continued on next page

Table 61 – continued from previous page

size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(np.ndarray) Internal state of the neurons in this layer [N,]
t	(float) The current evolution time of this layer
tau_syn	(np.ndarray) Synaptic time constants for this layer [N,]
weights	(ndarray) Weights encapsulated by this layer (MxN)
`xtx`	
`xty`	

Methods

__init__([weights, bias, dt, noise_std, ...])	Construct an exponential synapse layer (spiking input, pytorch as backend)
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input
load_from_dict(config)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save the contents of a dictionary to a file :param config: :param filename:
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the essential parameters of this layer to a dictionary for saving
train(ts_target, ts_input, is_first, is_last)	Wrapper to standardize training syntax across layers.
train_logreg(ts_target[, ts_input, ...])	Train self with logistic regression over one of possibly many batches.
train_rr(ts_target[, ts_input, regularize, ...])	train_rr - Train self with ridge regression over one of possibly many batches.

__init__(*weights: Union[numpy.ndarray, int] = None*, *bias: numpy.ndarray = 0*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_syn: float = 0.005*, *name: str = 'unnamed'*, *add_events: bool = True*, *max_num_timesteps: int = 5000*)

Construct an exponential synapse layer (spiking input, pytorch as backend)

Parameters

- **weights** – np.array MxN weight matrix int Size of layer -> creates one-to-one conversion layer

- **dt** – float Time step for state evolution
- **noise_std** – float Std. dev. of noise added to this layer. Default: 0
- **tau_syn** – float Output synaptic time constants. Default: 5ms
- **synapse_eq** – Brian2.Equations set of synapse equations for receiver. Default: exponential
- **integrator_name** – str Integrator to use for simulation. Default: 'exact'
- **name** – str Name for the layer. Default: 'unnamed'

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one.

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_batch_update(*inp: numpy.ndarray*, *target: numpy.ndarray*, *reset: bool*, *train_biases: bool*, *standardize: bool*, *update_weights: bool*, *return_training_progress: bool*) → Dict
Train with the already processed input and target data of the current batch. Update layer weights and biases if requested. Provide information on training state if requested.

Parameters

- **inp** (*np.ndarray*) – 2D-array (num_samples x num_features) of input data.
- **target** (*np.ndarray*) – 2D-array (num_samples x 'self.size') of target data.
- **reset** (*bool*) – If 'True', internal variables will be reset at the end.
- **train_bises** (*bool*) – Should biases be trained or only weights?
- **standardize** (*bool*) – Has input data been z-score standardized?
- **update_weights** (*bool*) – Set 'True' to update layer weights and biases.
- **return_training_progress** (*bool*) – Return intermediate training data (e.g. xtx, xty,...)

Return dict Dict with information on training progress, depending on values of other function arguments.

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_filter_data(*data: numpy.ndarray*, *num_timesteps: int*)

Filter input data y convolving with the synaptic kernel

Parameters

- **data** (*np.ndarray*) – Input data
- **num_timesteps** (*int*) – The number of time steps to return

Return np.ndarray The filtered data

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_gradients(*ct_weights*, *ct_biases*, *ct_input*, *ct_target*, *regularize*)

Compute weight gradients

Parameters

- **ct_weights** –
- **ct_biases** –
- **ct_input** –
- **ct_target** –

- **regularize** –

Returns

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>)*

Sample input and return as raster

Parameters

- **ts_input** (*Optional[TSEvent]*) – Spiking input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps

Return (spike_raster, num_timesteps) spike_raster: (np.ndarray) Raster containing spike info num_timesteps: (np.ndarray) Number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>)*

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_training_data(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *is_first: Optional[bool] = True*, *is_last: Optional[bool] = False*)

Check and rasterize input and target signals for this batch

Parameters

- **ts_target** (*TSContinuous*) – Target signal for this batch
- **ts_input** (*Optional[TSEvent]*) – Input signal for this batch. Default: `None`, no input for this batch
- **is_first** (*Optional[bool]*) – If `True`, this is the first batch in training. Default: `True`, this is the first batch
- **is_last** (*optional[bool]*) – If `True`, this is the last training batch. Default: `False`, this is not the last batch

:return (inp, target, time_base) inp np.ndarray: Rasterized input signal [T, M] target np.ndarray: Rasterized target signal [T, O] time_base np.ndarray: Time base for `inp` and `target`

_single_batch_evolution(*weighted_input: numpy.ndarray*, *num_timesteps: int*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input for a single batch

Parameters

- **weighted_input** – np.ndarray Weighted input
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

_update_kernels()
Generate kernels for filtering input spikes during evolution and training

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSContinuous
Function to evolve the states of this layer given an input

Parameters

- **tsSpkInput** – TSEvent Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSContinuous output spike series

property input_type
(*TSEvent*) Time series class accepted by this layer (*TSEvent*)

static load_from_dict(*config: dict*)
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file(*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property max_num_timesteps
(int) Maximum number of timesteps used by the synaptic kernel

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save the contents of a dictionary to a file :param config: :param filename:

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(np.ndarray) Internal state of the neurons in this layer [N,]

property t
(float) The current evolution time of this layer

property tau_syn
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict() → dict
Convert the essential parameters of this layer to a dictionary for saving

Return dict

train(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSContinuous*, *is_first: bool*, *is_last: bool*, *method: str = 'rr'*, ***kwargs*)

Wrapper to standardize training syntax across layers. Use specified training method to train layer for current batch.

Parameters

- **ts_target** – Target time series for current batch.
- **ts_input** – Input to the layer during the current batch.
- **is_first** – Set `True` to indicate that this batch is the first in training procedure.
- **is_last** – Set `True` to indicate that this batch is the last in training procedure.
- **method** – String indicating which training method to choose. Currently only ridge regression ("rr") and logistic regression are supported.
- **kwargs** – Will be passed on to corresponding training method.

train_logreg(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSEvent = None*, *learning_rate: float = 0*, *regularize: float = 0*, *batch_size: Optional[int] = None*, *epochs: int = 1*, *store_states: bool = True*, *verbose: bool = False*)

Train self with logistic regression over one of possibly many batches. Note that this training method assumes that a sigmoid funciton is applied to the layer output, which is not the case in `evolve`. Use pytorch as backend.

Parameters

- **ts_target** – TimeSeries - target for current batch
- **ts_input** – TimeSeries - input to self for current batch
- **learning_rate** – flaot - Factor determining scale of weight increments at each step
- **regularize** – float - regularization parameter
- **batch_size** – int - Number of samples per batch. If None, train with all samples at once
- **epochs** – int - How many times is training repeated
- **store_states** – bool - Include last state from previous training and store state from this traning. This has the same effect as if data from both trainings were presented at once.
- **verbose** – bool - Print output about training progress

train_rr(*ts_target: rockpool.timeseries.TSContinuous*, *ts_input: rockpool.timeseries.TSEvent = None*, *regularize: float = 0*, *is_first: bool = True*, *is_last: bool = False*, *store_states: bool = True*, *train_biases: bool = True*, *calc_intermediate_results: bool = False*, *return_training_progress: bool = False*) → Optional[Dict]

train_rr - Train self with ridge regression over one of possibly many batches. Use Kahan summation to reduce rounding errors when adding data to existing matrices from previous batches.

Parameters

- **ts_target** – TimeSeries - target for current batch
- **ts_input** – TimeSeries - input to self for current batch
- **train_biases** – bool - If True, train biases as if they were weights Otherwise present biases will be ignored in training and not be changed.
- **calc_intermediate_results** – bool - If True, calculates the intermediate weights not in the final batch

- **return_training_progress** – bool - If True, return dict of current training variables for each batch.

Regularize float - regularization for ridge regression

Is_first bool - True if current batch is the first in training

Is_last bool - True if current batch is the last in training

Store_states bool - Include last state from previous training and store state from this traning. This has the same effect as if data from both trainings were presented at once.

Returns If `return_training_progress`, return dict with current trainig variables (xtx, xty, kahan_comp_xtx, kahan_comp_xty). Weights and biases are returned if `is_last` or if `calc_intermediate_results`.

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.23 API reference for layers.FFIAFTorch

class layers.**FFIAFTorch**(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)
Bases: `rockpool.layers.layer.Layer`

Define a spiking feedforward layer with spiking outputs, with a PyTorch backend

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)
Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions
- **max_num_timesteps** – int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
t	(float) The current evolution time of this layer
tau_mem	
v_reset	
v_rest	
v_thresh	
weights	

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions
- **max_num_timesteps** – int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: rockpool.timeseries.TSContinuous = None*, *duration: float = None*, *num_timesteps: int = None) -> (<class 'torch.Tensor'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (`TSContinuous`) – TxM or Tx1 Input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme

- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Prepare the weighted, noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters **inp** – np.ndarray Input to layer as matrix

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

evolve : Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return TSEvent: output spike series

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

static load_from_dict(*config*)

load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)

load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std

(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.24 API reference for layers.FFIAFRefrTorch

class layers.**FFIAFRefrTorch**(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Bases: `rockpool.layers.gpl.pytorch.iaf_torch._RefractoryBase`, `rockpool.layers.gpl.pytorch.iaf_torch.FFIAFTorch`

A spiking feedforward layer with spiking outputs and refractoriness

__init__(*weights: numpy.ndarray, bias: Union[float, numpy.ndarray] = 0.015, dt: float = 0.0001, noise_std: float = 0, tau_mem: Union[float, numpy.ndarray] = 0.02, v_thresh: Union[float, numpy.ndarray] = -0.055, v_reset: Union[float, numpy.ndarray] = -0.065, v_rest: Union[float, numpy.ndarray] = -0.065, refractory=0, name: str = 'unnamed', record: bool = False, max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events. Supports Refractoriness.

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer name attribute
state	
synapse_state	
t	(float) The current evolution time of this layer

Continued on next page

Table 65 – continued from previous page

t_refr_countdown
tau_mem
v_reset
v_rest
v_thresh
weights

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events. Supports Refractoriness.

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV

- **refractory** – float Refractory period after each spike. Default: 0ms
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: rockpool.timeseries.TSContinuous = None*, *duration: float = None*, *num_timesteps: int = None) -> (<class 'torch.Tensor'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (`TSContinuous`) – TxM or Tx1 Input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Prepare the weighted, noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters **inp** – np.ndarray Input to layer as matrix

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

static load_from_dict(*config*)
load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.25 API reference for layers.FFIAFSpkInTorch

class layers.**FFIAFSpkInTorch**(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.01*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: numpy.ndarray = 0.02*, *tau_syn: numpy.ndarray = 0.02*, *v_thresh: numpy.ndarray = -0.055*, *v_reset: numpy.ndarray = -0.065*, *v_rest: numpy.ndarray = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)
Bases: `rockpool.layers.gpl.pytorch.iaf_torch.FFIAFTorch`

Spiking feedforward layer with spiking in- and outputs

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.01*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: numpy.ndarray = 0.02*, *tau_syn: numpy.ndarray = 0.02*, *v_thresh: numpy.ndarray = -0.055*, *v_reset: numpy.ndarray = -0.065*, *v_rest: numpy.ndarray = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)
Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. In- and outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV

- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Time series class accepted by this layer (*TSEvent*)
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer name attribute
state	
synapse_state	
t	(float) The current evolution time of this layer
tau_mem	
tau_syn	
v_reset	
v_rest	
v_thresh	
weights	

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer

Continued on next page

Table 68 – continued from previous page

reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.01*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: numpy.ndarray = 0.02*, *tau_syn: numpy.ndarray = 0.02*, *v_thresh: numpy.ndarray = -0.055*, *v_reset: numpy.ndarray = -0.065*, *v_rest: numpy.ndarray = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. In- and outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (`TSEvent`) – TxM or Tx1 spiking input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds

:param num_timesteps int Number of evolution time steps

Returns spike_raster: ndarray Boolean raster containing spike info num_timesteps: ndarray Number of evlution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Prepare the weighted, noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters **inp** – np.ndarray Input data

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

evolve : Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return TSEvent: output spike series

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(*TSEvent*) Time series class accepted by this layer (*TSEvent*)

static load_from_dict(*config*)
load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.26 API reference for layers.FFIAFSpkInRefrTorch

class layers.**FFIAFSpkInRefrTorch**(*weights: numpy.ndarray, bias: numpy.ndarray = 0.01, dt: float = 0.0001, noise_std: float = 0, tau_mem: numpy.ndarray = 0.02, tau_syn: numpy.ndarray = 0.02, v_thresh: numpy.ndarray = -0.055, v_reset: numpy.ndarray = -0.065, v_rest: numpy.ndarray = -0.065, refractory=0, name: str = 'unnamed', record: bool = False, max_num_timesteps: int = 400*)

Bases: rockpool.layers.gpl.pytorch.iaf_torch._RefractoryBase, rockpool.layers.gpl.pytorch.iaf_torch.FFIAFSpkInTorch

Spiking feedforward layer with spiking in- and outputs and refractoriness, using a PyTorch backend

__init__(*weights: numpy.ndarray, bias: numpy.ndarray = 0.01, dt: float = 0.0001, noise_std: float = 0, tau_mem: numpy.ndarray = 0.02, tau_syn: numpy.ndarray = 0.02, v_thresh: numpy.ndarray = -0.055, v_reset: numpy.ndarray = -0.065, v_rest: numpy.ndarray = -0.065, refractory=0, name: str = 'unnamed', record: bool = False, max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. In- and outputs are spiking events. Supports refractoriness.

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of self
dt	(float) Simulation time step of this layer
input_type	(*TSEvent*) Time series class accepted by this layer (*TSEvent*)
noise_std	(float) Noise injected into the state of this layer during evolution

Continued on next page

Table 69 – continued from previous page

output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
t	(float) The current evolution time of this layer
t_refr_countdown	
tau_mem	
tau_syn	
v_reset	
v_rest	
v_thresh	
weights	

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: numpy.ndarray = 0.01*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: numpy.ndarray = 0.02*, *tau_syn: numpy.ndarray = 0.02*, *v_thresh: numpy.ndarray = -0.055*, *v_reset: numpy.ndarray = -0.065*, *v_rest: numpy.ndarray = -0.065*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking feedforward layer with IAF neurons, running on GPU, using torch. In- and outputs are spiking events. Supports refractoriness.

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 10mA
- **dt** – float Time-step. Default: 0.1 ms
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 20ms
- **tau_syn** – np.array Nx1 vector of synapse time constants. Default: 20ms
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -55mV
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -65mV
- **refractory** – float Refractory period after each spike. Default: 0ms
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (`TSEvent`) – TxM or Tx1 spiking input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds

:param num_timesteps int Number of evolution time steps

Returns spike_raster: ndarray Boolean raster containing spike info num_timesteps: ndarray Number of evlution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Prepare the weighted, noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters **inp** – np.ndarray Input data

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(*TSEvent*) Time series class accepted by this layer (*TSEvent*)

static load_from_dict(*config*)
load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.27 API reference for layers.RecIAFTorch

class layers.**RecIAFTorch**(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_r: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)
Bases: `rockpool.layers.gpl.pytorch.iaf_torch.FFIAFTorch`

A spiking recurrent layer with input currents and spiking outputs

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_r: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.015
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 0.005
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Time step used by this layer, in s
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
t	(float) The current evolution time of this layer
tau_mem	
tau_syn_r	(np.ndarray) Synaptic time constants for this layer [N,]
v_reset	
v_rest	

Continued on next page

Table 71 – continued from previous page

v_thresh
weights

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from to_dict and save in a json file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_r: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.015
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 0.005
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>)*
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds

- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: rockpool.timeseries.TSContinuous = None*, *duration: float = None*, *num_timesteps: int = None) -> (<class 'torch.Tensor'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (`TSContinuous`) – TxM or Tx1 Input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)

Prepare the noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters

- **tsSpkInput** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property class_name
(str) Class name of `self`

property dt
(float) Time step used by this layer, in s

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

static load_from_dict(*config*)
load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

property tau_syn_r
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.28 API reference for layers.RecIAFRefrTorch

class layers.**RecIAFRefrTorch**(*weights: numpy.ndarray, bias: Union[float, numpy.ndarray] = 0.015, dt: float = 0.0001, noise_std: float = 0, tau_mem: Union[float, numpy.ndarray] = 0.02, tau_syn_r: Union[float, numpy.ndarray] = 0.05, v_thresh: Union[float, numpy.ndarray] = -0.055, v_reset: Union[float, numpy.ndarray] = -0.065, v_rest: Union[float, numpy.ndarray] = -0.065, refractory=0, name: str = 'unnamed', record: bool = False, max_num_timesteps: int = 400*)
Bases: `rockpool.layers.gpl.pytorch.iaf_torch._RefractoryBase`, `rockpool.layers.gpl.pytorch.iaf_torch.RecIAFTorch`

A spiking recurrent layer with current inputs, spiking outputs and refractoriness. PyTorch backend.

__init__(*weights: numpy.ndarray, bias: Union[float, numpy.ndarray] = 0.015, dt: float = 0.0001, noise_std: float = 0, tau_mem: Union[float, numpy.ndarray] = 0.02, tau_syn_r: Union[float, numpy.ndarray] = 0.05, v_thresh: Union[float, numpy.ndarray] = -0.055, v_reset: Union[float, numpy.ndarray] = -0.065, v_rest: Union[float, numpy.ndarray] = -0.065, refractory=0, name: str = 'unnamed', record: bool = False, max_num_timesteps: int = 400*)
Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events. Supports refractoriness

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.015
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02

- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 0.005
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **refractory** – float Refractory period after each spike. Default: 0
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Time step used by this layer, in s
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
t	(float) The current evolution time of this layer
t_refr_countdown	
tau_mem	
tau_syn_r	(np.ndarray) Synaptic time constants for this layer [N,]
v_reset	
v_rest	
v_thresh	
weights	

Methods

__init__(weights[, bias, dt, noise_std, ...])	Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch.

Continued on next page

Table 74 – continued from previous page

evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load_from_dict(config)	load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

__init__(*weights: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.015*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_r: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs are continuous currents; outputs are spiking events. Supports refractoriness

Parameters

- **weights** – np.array MxN weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.015
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_r** – np.array NxN vector of recurrent synaptic time constants. Default: 0.005
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **refractory** – float Refractory period after each spike. Default: 0
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None*) -> (<*class 'numpy.ndarray'*>, <*class 'int'*>)

Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If True, then None is permitted as argument for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray inp, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: rockpool.timeseries.TSContinuous = None*, *duration: float = None*, *num_timesteps: int = None) -> (<class 'torch.Tensor'>*, *<class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (TSContinuous) – TxM or Tx1 Input signals for this layer
- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a TSEvent time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either num_timesteps or the duration of ts_input will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of .dt. If not provided, then either duration or the duration of ts_input will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of .dt

_prepare_neural_input (*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Prepare the noisy synaptic input to the neurons and return it together with number of evolution time steps

Parameters

- **tsSpkInput** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time

:param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution (*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

property class_name

(str) Class name of self

property dt

(float) Time step used by this layer, in s

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

static load_from_dict(*config*)
load the layer from a dict :param config: dict information for the initialization :return: FFIAFTorch layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict*, *filename*)
Save a dictionary to a file

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

property tau_syn_r
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict()
Convert parameters of this layer to a dict if they are relevant for reconstructing an identical layer

Return Dict A dictionary that can be used to reconstruct the layer

12.4.29 API reference for layers.RecIAFSpkInTorch

class layers.**RecIAFSpkInTorch**(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)
Bases: `rockpool.layers.gpl.pytorch.iaf_torch.RecIAFTorch`

A spiking recurrent layer with spiking in- and outputs

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)
Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055

- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Time step used by this layer, in s
input_type	(*TSEvent*) Input time series class for this layer (*TSEvent*)
max_num_timesteps	(int) Maxmimum number of timesteps used by synaptic kernel
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
synapse_state_inp	
t	(float) The current evolution time of this layer
tau_mem	
tau_syn_inp	
tau_syn_r	(np.ndarray) Synaptic time constants for this layer [N,]
tau_syn_rec	
v_reset	
v_rest	
v_thresh	
weights	(np.ndarray) Recurrent weights for this layer [N, N]
weights_in	
weights_rec	

Methods

__init__(weights_in, weights_rec[, bias, ...])	Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load(filename)	Load parameters from a file and construct a layer from the parameters
load_from_dict(config)	Construct a layer from a dictionary of parameters
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset internal state and clock for this layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file in JSON format
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the essential parameters of this layer to a dictionary for saving

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input, set up time base

Parameters

- **ts_input** (`TSEvent`) – TxM or Tx1 Input spikes for this layer
- **duration** – float Duration of the desired evolution, in seconds

:param num_timesteps int Number of evolution time steps

Returns spike_raster: Tensor Boolean raster containing spike info num_timesteps: ndarray Number of evlution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None*) -> (<class 'numpy.ndarray'>, <class 'int'>)

Prepare the noisy synaptic input to the neurons and return it together with number of evolution time steps

:param inp np.ndarray External input spike raster :param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

_update_rec_kernel()
Update the kernel for this layer

property _weights
(np.ndarray) Recurrent weights for this layer [N, N]

property class_name
(str) Class name of `self`

property dt
(float) Time step used by this layer, in s

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(*TSEvent*) Input time series class for this layer (*TSEvent*)

static load(*filename: str*)
Load parameters from a file and construct a layer from the parameters

Parameters **filename** (*str*) – File to load from

Return RecIAFSpkInTorch Reconstructed layer

static load_from_dict(*config: dict*)
Construct a layer from a dictionary of parameters

Parameters **config** (*dict*) – Dictionary of parameters for the layer

Return RecIAFSpkInTorch Reconstructed layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property max_num_timesteps
(int) Maxmimum number of timesteps used by synaptic kernel

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset internal state and clock for this layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict: dict*, *filename: str*)
Save a dictionary to a file in JSON format

Parameters

- **essential_dict** (*dict*) – Dictionary to save
- **filename** (*str*) – File to save to

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

property tau_syn_r
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict()
Convert the essential parameters of this layer to a dictionary for saving

Return dict

property weights
(np.ndarray) Recurrent weights for this layer [N, N]

12.4.30 API reference for layers.RecIAFSpkInRefrTorch

class layers.**RecIAFSpkInRefrTorch**(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory: float = 0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Bases: rockpool.layers.gpl.pytorch.iaf_torch._RefractoryBase, rockpool.layers.gpl.pytorch.iaf_torch.RecIAFSpkInTorch

A spiking recurrent layer with spiking in- and outputs and refractoriness, and a PyTorch backend

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory: float = 0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events. Supports refractoriness

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **refractory** – float Refractory period after each spike. Default: 0
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Time step used by this layer, in s
input_type	(*TSEvent*) Input time series class for this layer (*TSEvent*)
max_num_timesteps	(int) Maxmimum number of timesteps used by synaptic kernel
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
synapse_state	
synapse_state_inp	
t	(float) The current evolution time of this layer
t_refr_countdown	
tau_mem	
tau_syn_inp	
tau_syn_r	(np.ndarray) Synaptic time constants for this layer [N,]
tau_syn_rec	
v_reset	
v_rest	
v_thresh	
weights	(np.ndarray) Recurrent weights for this layer [N, N]
weights_in	
weights_rec	

Methods

__init__(weights_in, weights_rec[, bias, ...])	Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load(filename)	Load parameters from a file and construct a layer from the parameters
load_from_dict(config)	Construct a layer from a dictionary of parameters

Continued on next page

Table 78 – continued from previous page

load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset internal state and clock for this layer
reset_state()	Reset the internal state of the layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file in JSON format
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the essential parameters of this layer to a dictionary for saving

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *noise_std: float = 0*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray] = -0.065*, *refractory: float = 0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events. Supports refractoriness

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **noise_std** – float Noise std. dev. per second. Default: 0
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron thresholds. Default: -0.065
- **refractory** – float Refractory period after each spike. Default: 0
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"

- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input, set up time base

Parameters

- **ts_input** (TSEvent) – TxM or Tx1 Input spikes for this layer
- **duration** – float Duration of the desired evolution, in seconds

:param num_timesteps int Number of evolution time steps

Returns spike_raster: Tensor Boolean raster containing spike info num_timesteps: ndarray Number of evlution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Prepare the noisy synaptic input to the neurons and return it together with number of evolution time steps

:param inp np.ndarray External input spike raster :param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent

Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – Input to layer as matrix

:param evolution_timestep Time step within current evolution at beginning of current batch :param num_timesteps: Number of evolution time steps :param verbose: Currently no effect, just for conformity :return: output spike series

_update_rec_kernel()

Update the kernel for this layer

property _weights

(np.ndarray) Recurrent weights for this layer [N, N]

property class_name
(str) Class name of `self`

property dt
(float) Time step used by this layer, in s

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(*TSEvent*) Input time series class for this layer (*TSEvent*)

static load(*filename: str*)
Load parameters from a file and construct a layer from the parameters

Parameters **filename** (*str*) – File to load from

Return RecIAFSpkInTorch Reconstructed layer

static load_from_dict(*config: dict*)
Construct a layer from a dictionary of parameters

Parameters **config** (*dict*) – Dictionary of parameters for the layer

Return RecIAFSpkInTorch Reconstructed layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property max_num_timesteps
(int) Maxmimum number of timesteps used by synaptic kernel

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset internal state and clock for this layer

reset_state()
Reset the internal state of the layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict: dict*, *filename: str*)
Save a dictionary to a file in JSON format

Parameters

- **essential_dict** (*dict*) – Dictionary to save
- **filename** (*str*) – File to save to

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

property tau_syn_r
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict()
Convert the essential parameters of this layer to a dictionary for saving

Return dict

property weights
(np.ndarray) Recurrent weights for this layer [N, N]

12.4.31 API reference for layers.RecIAFSpkInRefrCLTorch

class layers.**RecIAFSpkInRefrCLTorch**(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *leak_rate: Union[float, numpy.ndarray] = 0.02*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray, None] = -0.065*, *state_min: Union[float, numpy.ndarray, None] = -0.085*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Bases: rockpool.layers.gpl.pytorch.iaf_torch.RecIAFSpkInRefrTorch

A recurrent spiking layer with constant leak. Spiking inputs and outputs, PyTorch backend.

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *leak_rate: Union[float, numpy.ndarray] = 0.02*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray, None] = -0.065*, *state_min: Union[float, numpy.ndarray, None] = -0.085*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events. Support refractoriness. Constant leak.

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **leak_rate** – np.array Nx1 vector of constant neuron leakage in V/s. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron reset potential. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron resting potential. Default: -0.065 If None, leak will always be negative (for positive entries of leak_rate)
- **state_min** – np.array Nx1 vector of lower limits for neuron states. Default: -0.85 If None, there are no lower limits
- **refractory** – float Refractory period after each spike. Default: 0
- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

Attributes

alpha	
bias	
class_name	(str) Class name of `self`
dt	(float) Time step used by this layer, in s
input_type	(*TSEvent*) Input time series class for this layer (*TSEvent*)
leak_rate	
max_num_timesteps	(int) Maxmimum number of timesteps used by synaptic kernel
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
refractory	
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	
state_min	
synapse_state	
synapse_state_inp	
t	(float) The current evolution time of this layer
t_refr_countdown	
tau_mem	
tau_syn_inp	
tau_syn_r	(np.ndarray) Synaptic time constants for this layer [N,]
tau_syn_rec	
v_reset	
v_rest	
v_thresh	
weights	(np.ndarray) Recurrent weights for this layer [N, N]
weights_in	
weights_rec	

Methods

__init__(weights_in, weights_rec[, bias, ...])	Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch.
evolve([ts_input, duration, num_timesteps, ...])	Function to evolve the states of this layer given an input.
load(filename)	Load parameters from a file and construct a layer from the parameters
load_from_dict(config)	Construct a layer from a dictionary of parameters
load_from_file(filename)	load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset internal state and clock for this layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(essential_dict, filename)	Save a dictionary to a file in JSON format
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the essential parameters of this layer to a dictionary for saving

__init__(*weights_in: numpy.ndarray*, *weights_rec: numpy.ndarray*, *bias: Union[float, numpy.ndarray] = 0.0105*, *dt: float = 0.0001*, *leak_rate: Union[float, numpy.ndarray] = 0.02*, *tau_mem: Union[float, numpy.ndarray] = 0.02*, *tau_syn_inp: Union[float, numpy.ndarray] = 0.05*, *tau_syn_rec: Union[float, numpy.ndarray] = 0.05*, *v_thresh: Union[float, numpy.ndarray] = -0.055*, *v_reset: Union[float, numpy.ndarray] = -0.065*, *v_rest: Union[float, numpy.ndarray, None] = -0.065*, *state_min: Union[float, numpy.ndarray, None] = -0.085*, *refractory=0*, *name: str = 'unnamed'*, *record: bool = False*, *add_events: bool = True*, *max_num_timesteps: int = 400*)

Construct a spiking recurrent layer with IAF neurons, running on GPU, using torch. Inputs and outputs are spiking events. Support refractoriness. Constant leak.

Parameters

- **weights_in** – np.array MxN input weight matrix.
- **weights_rec** – np.array NxN recurrent weight matrix.
- **bias** – np.array Nx1 bias vector. Default: 0.0105
- **dt** – float Time-step. Default: 0.0001
- **tau_mem** – np.array Nx1 vector of neuron time constants. Default: 0.02
- **leak_rate** – np.array Nx1 vector of constant neuron leakage in V/s. Default: 0.02
- **tau_syn_inp** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **tau_syn_rec** – np.array Nx1 vector of synapse time constants. Default: 0.05
- **v_thresh** – np.array Nx1 vector of neuron thresholds. Default: -0.055
- **v_reset** – np.array Nx1 vector of neuron reset potential. Default: -0.065
- **v_rest** – np.array Nx1 vector of neuron resting potential. Default: -0.065 If None, leak will always be negative (for positive entries of leak_rate)
- **state_min** – np.array Nx1 vector of lower limits for neuron states. Default: -0.85 If None, there are no lower limits
- **refractory** – float Refractory period after each spike. Default: 0

- **name** – str Name for the layer. Default: 'unnamed'
- **record** – bool Record membrane potential during evolutions. Default: False

Add_events bool If during evolution multiple input events arrive during one time step for a channel, count their actual number instead of just counting them as one (This might make less sense for refractory neurons).

Max_num_timesteps int Maximum number of timesteps during single evolution batch. Longer evolution periods will automatically split in smaller batches.

_batch_data(*inp: numpy.ndarray*, *num_timesteps: int*, *max_num_timesteps: int = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)
Generator that returns the data in batches

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size

- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray
Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input, set up time base

Parameters

- **ts_input** (`TSEvent`) – TxM or Tx1 Input spikes for this layer
- **duration** – float Duration of the desired evolution, in seconds

:param num_timesteps int Number of evolution time steps

Returns spike_raster: Tensor Boolean raster containing spike info num_timesteps: ndarray Number of evlution time steps

_prepare_input_events(*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_neural_input(*inp: numpy.array*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>*, *<class 'int'>*)
Prepare the noisy synaptic input to the neurons and return it together with number of evolution time steps

:param inp np.ndarray External input spike raster :param num_timesteps int Number of evolution time steps :return:

neural_input np.ndarray Input to neurons num_timesteps int Number of evolution time steps

_single_batch_evolution(*inp: numpy.ndarray*, *evolution_timestep: int*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input for a single batch

Parameters **inp** – np.ndarray Input to layer as matrix

:param evolution_timestep int Time step within current evolution at beginning of current batch :param num_timesteps: int Number of evolution time steps :param verbose: bool Currently no effect, just for conformity :return: TSEvent output spike series

_update_rec_kernel()
Update the kernel for this layer

property _weights
(np.ndarray) Recurrent weights for this layer [N, N]

property class_name
(str) Class name of `self`

property dt
(float) Time step used by this layer, in s

evolve(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = False*) → rockpool.timeseries.TSEvent
Function to evolve the states of this layer given an input. Automatically splits evolution in batches

Parameters

- **ts_input** – TSContinuous Input spike trian
- **duration** – float Simulation/Evolution time
- **num_timesteps** – int Number of evolution time steps
- **verbose** – bool Currently no effect, just for conformity

Returns TSEvent output spike series

property input_type
(*TSEvent*) Input time series class for this layer (*TSEvent*)

static load(*filename: str*)
Load parameters from a file and construct a layer from the parameters

Parameters **filename** (*str*) – File to load from

Return RecIAFSpkInTorch Reconstructed layer

static load_from_dict(*config: dict*)
Construct a layer from a dictionary of parameters

Parameters **config** (*dict*) – Dictionary of parameters for the layer

Return RecIAFSpkInTorch Reconstructed layer

static load_from_file(*filename*)
load the layer from a file :param filename: str with the filename that includes the dict that initializes the layer :return: FFIAFTorch layer

property max_num_timesteps
(int) Maxmimum number of timesteps used by synaptic kernel

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset internal state and clock for this layer

reset_state()
Reset the internal state of this layer

reset_time()
Reset the internal clock of this layer to 0

save(*essential_dict: dict*, *filename: str*)
Save a dictionary to a file in JSON format

Parameters

- **essential_dict** (*dict*) – Dictionary to save
- **filename** (*str*) – File to save to

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property t
(float) The current evolution time of this layer

property tau_syn_r
(np.ndarray) Synaptic time constants for this layer [N,]

to_dict()
Convert the essential parameters of this layer to a dictionary for saving

Return dict

property weights
(np.ndarray) Recurrent weights for this layer [N, N]

12.4.32 API reference for layers.RecDynapSE

class layers.**RecDynapSE**(*weights_in: numpy.ndarray, weights_rec: numpy.ndarray, neuron_ids: Optional[numpy.ndarray] = None, virtual_neuron_ids: Optional[numpy.ndarray] = None, dt: Optional[float] = 2e-05, max_num_trials_batch: Optional[float] = None, max_batch_dur: Optional[float] = None, max_num_timesteps: Optional[int] = None, max_num_events_batch: Optional[int] = None, l_input_core_ids: List[int] = [0], input_chip_id: int = 0, clearcores_list: Optional[list] = None, controller: rockpool.devices.dynapse_control_extd.DynapseControlExtd = None, rpyc_port: Optional[int] = None, name: Optional[str] = 'unnamed', skip_weights: bool = False, skip_neuron_allocation: bool = False, fastmode: bool = False, speedup: float = 1.0*)

Bases: rockpool.layers.layer.Layer

Recurrent spiking layer implemented with a DynapSE backend.

This class represents a recurrent layer of spiking neurons, implemented with a HW backend on DynapSE hardware from aiCTX.

__init__(*weights_in: numpy.ndarray, weights_rec: numpy.ndarray, neuron_ids: Optional[numpy.ndarray] = None, virtual_neuron_ids: Optional[numpy.ndarray] = None, dt: Optional[float] = 2e-05, max_num_trials_batch: Optional[float] = None, max_batch_dur: Optional[float] = None, max_num_timesteps: Optional[int] = None, max_num_events_batch: Optional[int] = None, l_input_core_ids: List[int] = [0], input_chip_id: int = 0, clearcores_list: Optional[list] = None, controller: rockpool.devices.dynapse_control_extd.DynapseControlExtd = None, rpyc_port: Optional[int] = None, name: Optional[str] = 'unnamed', skip_weights: bool = False, skip_neuron_allocation: bool = False, fastmode: bool = False, speedup: float = 1.0*)

Recurrent spiking layer implemented with a DynapSE backend

Parameters

- **weights_in** (*ndarray[int]*) – MxN matrix of input weights from virtual to hardware neurons. Supplied in units of synaptic connection.
- **weights_rec** (*ndarray[int]*) – NxN matrix of weights between hardware neurons. Supplied in units of synaptic connection. Negative elements give rise to inhibitory synapses.
- **neuron_ids** (*Optional[ndarray[int]]*) – 1xN array of IDs of hardware neurons that are to be used as layer neurons. Default: None
- **virtual_neuron_ids** (*Optional[ndarray[int]]*) – 1xM array of IDs of virtual neurons that are to be used as input neurons. Default: None
- **dt** (*Optional[float]*) – Layer time-step. Default: 2e-5 s
- **max_num_trials_batch** (*Optional[int]*) – Maximum number of trials (specified in input timeseries) per batch.Longer evolution periods will automatically split in smaller batches.
- **max_batch_dur** (*Optional[float]*) – Maximum duration of single evolution batch.
- **max_num_timesteps** (*Optional[int]*) – Maximum number of time steps of of single evolution batch.
- **max_num_events_batch** (*Optional[float]*) – Maximum number of input events per evolution batch.

- **l_input_core_ids** (*Optional[ArrayLike[int]]*) – IDs of the cores that contain neurons receiving external inputs. To avoid ID collisions neurons on these cores should not receive inputs from other neurons.
- **input_chip_id** (*Optional[int]*) – ID of the chip with neurons that receive external input.
- **clearcores_list** (*Optional[List[int]]*) – IDs of chips where configurations should be cleared.
- **controller** (*Optional[DynapseControl]*) – DynapseControl object to use to interface with the DynapSE hardware.
- **rpyc_port** (*Optional[int]*) – Port at which RPyC connection should be established. Only considered if controller is `None`.
- **name** (*Optional[str]*) – Layer name.
- **skip_weights** (*Optional[bool]*) – Do not upload weight configuration to chip. (Use carefully)
- **skip_neuron_allocation** (*Optional[bool]*) – If `True`, do not verify if neurons are usable. Default: `False` (check that specified neurons are usable).
- **fastmode** (*Optional[bool]*) – If `True`, `DynapseControl` will not load buffered event filters when data is sent. Recording buffer is set to 0. (No effect with `RecDynapSEDemo` class). Default:`False`.
- **speedup** (*Optional[float]*) – If `fastmode`==True`, speed up input events to Dynapse by this factor. (No effect with `RecDynapSEDemo` class). Default: 1.0 (no speedup)

Attributes

class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(TimeSeries subclass) The input class of this layer (`TSEvent`)
l_input_core_ids	Core mask as a reversed binary string :return:
max_batch_dur	(int) Maximum duration of a batch, in integer timesteps.
max_num_events_batch	(int) Maximum number of events in a batch.
max_num_timesteps	(int) Maxmimum number of timesteps.
max_num_trials_batch	(int) Maximum number of trials in a batch.
neuron_ids	(ndarray[int]) 1xN array of neuron IDs implementing the recurrent layer :return:
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(TimeSeries subclass) The ouput class of this layer (`TSEvent`)
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)

Continued on next page

Table 81 – continued from previous page

start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
virtual_neuron_ids	(ndarray[int]) 1xM Array of virtual neuron IDs implementing the input layer :return:
weights	(ndarray[float]) NxN array of recurrent weights.
weights_in	(ndarray[float]) MxN array of input weights from input neurons to recurrent neurons
weights_rec	(ndarray[float]) NxN array of recurrent weights :return:

Methods

__init__(weights_in, weights_rec[, ...])	Recurrent spiking layer implemented with a DynapSE backend
evolve([ts_input, duration, num_timesteps, ...])	Evolve the layer by queueing spikes, stimulating and recording
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Return a dictionary encapsulating the layer

__init__(*weights_in: numpy.ndarray, weights_rec: numpy.ndarray, neuron_ids: Optional[numpy.ndarray] = None, virtual_neuron_ids: Optional[numpy.ndarray] = None, dt: Optional[float] = 2e-05, max_num_trials_batch: Optional[float] = None, max_batch_dur: Optional[float] = None, max_num_timesteps: Optional[int] = None, max_num_events_batch: Optional[int] = None, l_input_core_ids: List[int] = [0], input_chip_id: int = 0, clearcores_list: Optional[list] = None, controller: rockpool.devices.dynapse_control_extd.DynapseControlExtd = None, rpyc_port: Optional[int] = None, name: Optional[str] = 'unnamed', skip_weights: bool = False, skip_neuron_allocation: bool = False, fastmode: bool = False, speedup: float = 1.0*)

Recurrent spiking layer implemented with a DynapSE backend

Parameters

- **weights_in** (*ndarray[int]*) – MxN matrix of input weights from virtual to hardware neurons. Supplied in units of synaptic connection.
- **weights_rec** (*ndarray[int]*) – NxN matrix of weights between hardware neurons. Supplied in units of synaptic connection. Negative elements give rise to inhibitory synapses.
- **neuron_ids** (*Optional[ndarray[int]]*) – 1xN array of IDs of hardware neu-

rons that are to be used as layer neurons. Default: None

- **virtual_neuron_ids** (*Optional[ndarray[int]]*) – 1xM array of IDs of virtual neurons that are to be used as input neurons. Default: None
- **dt** (*Optional[float]*) – Layer time-step. Default: 2e-5 s
- **max_num_trials_batch** (*Optional[int]*) – Maximum number of trials (specified in input timeseries) per batch.Longer evolution periods will automatically split in smaller batches.
- **max_batch_dur** (*Optional[float]*) – Maximum duration of single evolution batch.
- **max_num_timesteps** (*Optional[int]*) – Maximum number of time steps of of single evolution batch.
- **max_num_events_batch** (*Optional[float]*) – Maximum number of input events per evolution batch.
- **l_input_core_ids** (*Optional[ArrayLike[int]]*) – IDs of the cores that contain neurons receiving external inputs. To avoid ID collisions neurons on these cores should not receive inputs from other neurons.
- **input_chip_id** (*Optional[int]*) – ID of the chip with neurons that receive external input.
- **clearcores_list** (*Optional[List[int]]*) – IDs of chips where configurations should be cleared.
- **controller** (*Optional[DynapseControl]*) – DynapseControl object to use to interface with the DynapSE hardware.
- **rpyc_port** (*Optional[int]*) – Port at which RPyC connection should be established. Only considered if controller is `None`.
- **name** (*Optional[str]*) – Layer name.
- **skip_weights** (*Optional[bool]*) – Do not upload weight configuration to chip. (Use carefully)
- **skip_neuron_allocation** (*Optional[bool]*) – If `True`, do not verify if neurons are usable. Default: `False` (check that specified neurons are usable).
- **fastmode** (*Optional[bool]*) – If `True`, `DynapseControl` will not load buffered event filters when data is sent. Recording buffer is set to 0. (No effect with `RecDynapSEDemo` class). Default:`False`.
- **speedup** (*Optional[float]*) – If `fastmode`==True, speed up input events to Dynapse by this factor. (No effect with `RecDynapSEDemo` class). Default: 1.0 (no speedup)

_batch_input_data(*ts_input: rockpool.timeseries.TSEvent*, *num_timesteps: int*, *verbose: bool = False*) -> (<*class 'numpy.ndarray'*>, <*class 'int'*>)

Generator that returns the data in batches

Parameters

- **ts_input** (`TSEvent`) – Input event time series of data to convert into batches
- **num_timesteps** (*int*) – Number of time steps to return in each batch
- **verbose** (*Optional[bool]*) – If `True`, display information about the batch. Default: `False`, do not display information

:yields : (vn_tpts_evts_inp_batch, vn_chnls_inp_batch, tstp_start_batch, num_tstps_batch * self.dt)

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray
Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray inp, possibly with dimensions repeated

_compile_weights_and_configure()
Configure DynapSE weights from the weight matrices

Use the input and recurrent weight matrices to determine an approximate discretised synapse configuration consistent with the weights, and try to configure the DynapSE hardware with the configuration.

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → int
Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, num_timesteps or the duration of ts_input will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of dt. If not provided, duration or the duration of ts_input will be used to determine evolution time

Return int Number of evolution time steps

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If True, allow None as a value for inp. Otherwise an error will be raised. Default: True, allow None

Return ndarray Values of inp, replicated out to the size of the current layer

Raises

- **AssertionError** – If inp is incompatibly sized to replicate out to the layer size
- **AssertionError** – If inp is None, and allow_none is False

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray
Replicate out a scalar to an array of shape shape

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to

- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds

- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input (*ts_input: Optional[rockpool.timeseries.TimeSeries] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'float'>*)

Sample input, set up time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TimeSeries]*) – `TimeSeries` of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will define the evolution time
- **num_timesteps** (*Optional[int]*) – Integer number of evolution time steps, in units of `.dt`. If not provided, then `duration` or the duration of `ts_input` will define the evolution time

Return (ndarray, ndarray, float) (time_base, input_steps, duration) time_base: T1 Discretised time base for evolution input_steps: (T1xN) Discretised input signal for layer num_timesteps: Actual number of evolution time steps, in units of `.dt`

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

_send_batch (*timesteps: numpy.ndarray*, *channels: numpy.ndarray*, *dur_batch: float*)

Send a batch of input events to the hardware

Parameters

- **timesteps** (*ndarray*) – 1xT array of event times
- **channels** (*ndarray*) – 1xT array of event channels corresponding to event times in `timesteps`
- **dur_batch** (*float*) – Duration of this batch in seconds

Return Tuple[times_out, channels_out] Spike data emitted by the hardware during this batch

property _weights
(ndarray[float]) NxN array of recurrent weights. Alias for `._weights_rec`.

property class_name
(str) Class name of `self`

property dt
(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*, *verbose: bool = True*) → rockpool.timeseries.TSEvent
Evolve the layer by queueing spikes, stimulating and recording

Parameters

- **ts_input** (*Optional[TSEvent]*) – input time series, containing `self.size` channels. Default: `None`, just record for a specified duration
- **duration** (*Optional[float]*) – Desired evolution duration, in seconds. Default: `None`, use the duration implied by `ts_input`
- **num_timesteps** (*Optional[int]*) – Desired evolution duration, in integer steps of `self.dt`. Default: `None`, use `duration` or ‘‘ts_input‘ to determine duration
- **verbose** (*Optional[bool]*) – If `True`, display information on evolution progress. Default: `True`, display information during evolution

Return TSEvent Output spikes emitted by the neurons in this layer, during the evolution time

property input_type
(TimeSeries subclass) The input class of this layer (`TSEvent`)

property l_input_core_ids
Core mask as a reversed binary string :return:

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls
Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property max_batch_dur
(int) Maximum duration of a batch, in integer timesteps. `None` or `int` > 0

property max_num_events_batch
(int) Maximum number of events in a batch. `int` >= 0

property max_num_timesteps
(int) Maxmimum number of timesteps. `None` or `int` > 0.

property max_num_trials_batch
(int) Maximum number of trials in a batch. `None` or `int` > 0.

property neuron_ids
(ndarray[int]) 1xN array of neuron IDs implementing the recurrent layer :return:

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(TimeSeries subclass) The ouput class of this layer (`TSEvent`)

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save(*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer(*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

to_dict()
Return a dictionary encapsulating the layer

Return Dict A description of the layer that can be used to recreate the object

property virtual_neuron_ids
(ndarray[int]) 1xM Array of virtual neuron IDs implementing the input layer :return:

property weights
(ndarray[float]) NxN array of recurrent weights. Alias for `.weights_rec`. :return:

property weights_in
(ndarray[float]) MxN array of input weights from input neurons to recurrent neurons

property weights_rec
(ndarray[float]) NxN array of recurrent weights :return:

12.4.33 API reference for layers.RecRateEulerJax

class layers.**RecRateEulerJax**(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: Optional[float] = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

Bases: `rockpool.layers.layer.Layer`

JAX-backed firing-rate recurrent layer

RecRateEuler implements a recurrent reservoir with input and output weighting, using a JAX-implemented solver as a back end. The design permits gradient-based learning of weights, biases and time constants using `jax.grad`.

RecRateEuler is compatible with the `layers.training.train_jax_sgd` module.

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: Optional[float] = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)
RecRateEulerJax - JAX-backed firing rate reservoir

Parameters

- **w_in** (*np.ndarray*) – Input weights [IxN]
- **w_recurrent** (*np.ndarray*) – Recurrent weights [NxN]

- **w_out** (*np.ndarray*) – Output weights [NxO]
- **tau** (*np.ndarray*) – Time constants [N]
- **bias** (*np.ndarray*) – Bias values [N]
- **noise_std** (*Optional[float]*) – White noise standard deviation applied to reservoir neurons. Default: `0.0`
- **Optional[Callable]activation_func** – Neuron transfer function f(x: float) -> float. Must be vectorised. Default: H_ReLU
- **dt** (*Optional[float]*) – Reservoir time step. Default: `np.min(tau) / 10.0`
- **name** (*Optional[str]*) – Name of the layer. Default: `None`
- **RNG key] rng_key Jax RNG key to use for noise. Default** (*Optional[Jax*) – Internally generated

Attributes

bias	
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau	
w_in	(np.ndarray) [IxN] input weights
w_out	(np.ndarray) [NxO] output weights
w_recurrent	(np.ndarray) [NxN] recurrent weights
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(w_in, w_recurrent, w_out, tau, bias)	RecRateEulerJax - `JAX`-backed firing rate reservoir
evolve([ts_input, duration, num_timesteps])	evolve() - Evolve the reservoir state
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file

Continued on next page

Table 84 – continued from previous page

randomize_state()	Randomize the internal state of this layer
reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a `json` file
save_layer(filename)	Obtain layer paramters from *to_dict* and save in a `json` file
to_dict()	Convert the parameters of this class to a dictionary

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_recurrent: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: Optional[float] = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

RecRateEulerJax - `JAX`-backed firing rate reservoir

Parameters

- **w_in** (*np.ndarray*) – Input weights [IxN]
- **w_recurrent** (*np.ndarray*) – Recurrent weights [NxN]
- **w_out** (*np.ndarray*) – Output weights [NxO]
- **tau** (*np.ndarray*) – Time constants [N]
- **bias** (*np.ndarray*) – Bias values [N]
- **noise_std** (*Optional[float]*) – White noise standard deviation applied to reservoir neurons. Default: `0.0`
- **Optional[Callable]activation_func** – Neuron transfer function f(x: float) -> float. Must be vectorised. Default: H_ReLU
- **dt** (*Optional[float]*) – Reservoir time step. Default: `np.min(tau) / 10.0`
- **name** (*Optional[str]*) – Name of the layer. Default: `None`
- **RNG key] rng_key Jax RNG key to use for noise. Default** (*Optional[Jax*) – Internally generated

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer

- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*inps: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]

_evolve_raw() - Raw evolution of an input array

Parameters **inps** – np.ndarray Input matrix [T, I]

Returns (res_inputs, rec_inputs, res_acts, outputs) res_inputs: np.ndarray Weighted inputs to reservoir units [T, N] rec_inputs np.ndarray Recurrent inputs to reservoir units [T, N] res_acts np.ndarray Reservoir activity trace [T, N] outputs np.ndarray Output of network [T, O]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'jax.numpy.lax_numpy.ndarray'>*, *<class 'jax.numpy.lax_numpy.ndarray'>*, *<class 'float'>*)

_prepare_input - Sample input, set up time base

Parameters

- **ts_input** – TimeSeries TxM or Tx1 Input signals for this layer

- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) time_base: ndarray T1 Discretised time base for evolution input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → rockpool.timeseries.TimeSeries

evolve() - Evolve the reservoir state

Parameters

- **ts_input** – TSContinuous Input time series
- **duration** – float Duration of evolution in seconds
- **num_timesteps** – int Number of time steps to evolve (based on self.dt)

Returns ts_output: TSContinuous Output time series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass

- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

to_dict() → dict
Convert the parameters of this class to a dictionary

Return dict

property w_in
(np.ndarray) [IxN] input weights

property w_out
(np.ndarray) [NxO] output weights

property w_recurrent
(np.ndarray) [NxN] recurrent weights

property weights
(ndarray) Weights encapsulated by this layer (MxN)

12.4.34 API reference for layers.ForceRateEulerJax

class layers.**ForceRateEulerJax**(*w_in: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: float = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)
Bases: `rockpool.layers.gpl.rate_jax.RecRateEulerJax`

Implements a pseudo recurrent reservoir, for use in reservoir transfer

In this layer, input and output weights are present, but no recurrent connectivity exists. Instead, "recurrent inputs" are injected into each layer neuron. The activations of the neurons are then compared to those of a target reservoir, and the recurrent weights can be solved for using linear regression.

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: float = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)
`JAX`-backed firing rate reservoir, used for reservoir transfer

Parameters

- **w_in** (*np.ndarray*) – Input weights [IxN]
- **w_out** (*np.ndarray*) – Output weights [NxO]

- **tau** (*np.ndarray*) – Time constants [N]
- **bias** (*np.ndarray*) – Bias values [N]
- **noise_std** (*Optional[float]*) – White noise standard deviation applied to reservoir neurons. Default: `0.0`
- **activation_func** (*Optional[Callable]*) – Neuron transfer function f(x: float) -> float. Must be vectorised. Default: `H_ReLU`
- **dt** (*Optional[float]*) – Reservoir time step. Default: `np.min(tau) / 10.0`
- **name** (*Optional[str]*) – Name of the layer. Default: `None`
- **RNG key] rng_key Jax RNG key to use for noise. Default** (*Optional[Jax*) – Internally generated

Attributes

bias	
class_name	(str) Class name of `self`
dt	(float) Simulation time step of this layer
input_type	(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.
noise_std	(float) Noise injected into the state of this layer during evolution
output_type	(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.
size	(int) Number of units in this layer (N)
size_in	(int) Number of input channels accepted by this layer (M)
size_out	(int) Number of output channels produced by this layer (O)
start_print	(str) Return a string containing the layer subclass name and the layer `name` attribute
state	(ndarray) Internal state of this layer (N)
t	(float) The current evolution time of this layer
tau	
w_in	(np.ndarray) [IxN] input weights
w_out	(np.ndarray) [NxO] output weights
w_recurrent	(np.ndarray) [NxN] recurrent weights
weights	(ndarray) Weights encapsulated by this layer (MxN)

Methods

__init__(w_in, w_out, tau, bias[, ...])	`JAX`-backed firing rate reservoir, used for reservoir transfer
evolve([ts_input, ts_force, duration, ...])	evolve() - Evolve the reservoir state
load_from_dict(config, **kwargs)	Generate instance of a *Layer* subclass with parameters loaded from a dictionary
load_from_file(filename, **kwargs)	Generate an instance of a *Layer* subclass, with parameters loaded from a file
randomize_state()	Randomize the internal state of this layer

Continued on next page

Table 86 – continued from previous page

reset_all()	Reset both the internal clock and the internal state of the layer
reset_state()	Reset the internal state of this layer
reset_time()	Reset the internal clock of this layer to 0
save(config, filename)	Save a set of parameters to a json file
save_layer(filename)	Obtain layer paramters from to_dict and save in a json file
to_dict()	Convert the layer to a dictionary for saving :return dict:

__init__(*w_in: jax.numpy.lax_numpy.ndarray, w_out: jax.numpy.lax_numpy.ndarray, tau: jax.numpy.lax_numpy.ndarray, bias: jax.numpy.lax_numpy.ndarray, noise_std: float = 0.0, activation_func: Optional[Callable[Union[float, jax.numpy.lax_numpy.ndarray], Union[float, jax.numpy.lax_numpy.ndarray]]] = <function H_ReLU>, dt: Optional[float] = None, name: Optional[str] = None, rng_key: Optional[int] = None*)

JAX-backed firing rate reservoir, used for reservoir transfer

Parameters

- **w_in** (*np.ndarray*) – Input weights [IxN]
- **w_out** (*np.ndarray*) – Output weights [NxO]
- **tau** (*np.ndarray*) – Time constants [N]
- **bias** (*np.ndarray*) – Bias values [N]
- **noise_std** (*Optional[float]*) – White noise standard deviation applied to reservoir neurons. Default: `0.0`
- **activation_func** (*Optional[Callable]*) – Neuron transfer function f(x: float) -> float. Must be vectorised. Default: `H_ReLU`
- **dt** (*Optional[float]*) – Reservoir time step. Default: `np.min(tau) / 10.0`
- **name** (*Optional[str]*) – Name of the layer. Default: `None`
- **RNG key] rng_key Jax RNG key to use for noise. Default** (*Optional[Jax*) – Internally generated

_check_input_dims(*inp: numpy.ndarray*) → numpy.ndarray

Verify if dimensions of an input matches this layer instance

If input dimension == 1, scale it up to self._size_in by repeating signal.

Parameters **inp** (*ndarray*) – ArrayLike containing input data

Return ndarray `inp`, possibly with dimensions repeated

_determine_timesteps(*ts_input: Optional[rockpool.timeseries.TimeSeries] = None, duration: Optional[float] = None, num_timesteps: Optional[int] = None*) → int

Determine how many time steps to evolve with the given input

Parameters

- **ts_input** (*Optional[TimeSeries]*) – TxM or Tx1 time series of input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, `num_timesteps` or the duration of `ts_input` will be used to determine evolution time

- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `dt`. If not provided, `duration` or the duration of `ts_input` will be used to determine evolution time

Return int Number of evolution time steps

_evolve_raw(*inps: jax.numpy.lax_numpy.ndarray*, *forces: jax.numpy.lax_numpy.ndarray*) → Tuple[jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray, jax.numpy.lax_numpy.ndarray]

_evolve_raw() - Raw evolution of an input array

Parameters

- **inps** – np.ndarray Input matrix [T, I]
- **forces** – np.ndarray Forcing signals [T, N]

Returns (res_inputs, rec_inputs, res_acts, outputs) res_inputs: np.ndarray Weighted inputs to forced reservoir units [T, N] res_acts np.ndarray Reservoir activity trace [T, N] outputs np.ndarray Output of network [T, O]

_expand_to_net_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_expand_to_shape(*inp*, *shape: tuple*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to an array of shape `shape`

Parameters

- **inp** (*Any*) – scalar or array-like of input data
- **shape** (*Tuple[int]*) – tuple defining array shape that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, then `None` is permitted as argument for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray `inp`, replicated to the correct shape

Raises

- **AssertionError** – If `inp` is shaped incompatibly to be replicated to the desired shape
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_size(*inp*, *size: int*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to a desired size

Parameters

- **inp** (*Any*) – scalar or array-like
- **size** (*int*) – Size that input should be expanded to
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optional[bool]*) – If `True`, allow None as a value for `inp`. Otherwise and error will be raised. Default: `True`, allow `None`

Return ndarray Array of `inp`, possibly expanded to the desired size

Raises

- **AssertionError** – If `inp` is incompatibly shaped to expand to the desired size
- **AssertionError** – If `inp` is `None` and `allow_none` is `False`

_expand_to_weight_size(*inp*, *var_name: str = 'input'*, *allow_none: bool = True*) → numpy.ndarray

Replicate out a scalar to the size of the layer's weights

Parameters

- **inp** (*Any*) – scalar or array-like
- **var_name** (*Optional[str]*) – Name of the variable to include in error messages. Default: "input"
- **allow_none** (*Optionbal[bool]*) – If `True`, allow `None` as a value for `inp`. Otherwise an error will be raised. Default: `True`, allow `None`

Return ndarray Values of `inp`, replicated out to the size of the current layer

Raises

- **AssertionError** – If `inp` is incompatibly sized to replicate out to the layer size
- **AssertionError** – If `inp` is `None`, and `allow_none` is `False`

_gen_time_trace(*t_start: float*, *num_timesteps: int*) → numpy.ndarray

Generate a time trace starting at t_start, of length num_timesteps+1 with time step length self._dt. Make sure it does not go beyond t_start+duration.

Parameters

- **t_start** (*float*) – Start time, in seconds
- **num_timesteps** (*int*) – Number of time steps to generate, in units of `.dt`

Return (ndarray) Generated time trace

_prepare_input(*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'jax.numpy.lax_numpy.ndarray'>*, *<class 'jax.numpy.lax_numpy.ndarray'>*, *<class 'float'>*)

_prepare_input - Sample input, set up time base

Parameters

- **ts_input** – TimeSeries TxM or Tx1 Input signals for this layer

- **duration** – float Duration of the desired evolution, in seconds
- **num_timesteps** – int Number of evolution time steps

Returns (time_base, input_steps, duration) time_base: ndarray T1 Discretised time base for evolution input_steps: ndarray (T1xN) Discretised input signal for layer num_timesteps: int Actual number of evolution time steps

_prepare_input_events (*ts_input: Optional[rockpool.timeseries.TSEvent] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None) -> (<class 'numpy.ndarray'>, <class 'int'>*)

Sample input from a `TSEvent` time series, set up evolution time base

This function checks an input signal, and prepares a discretised time base according to the time step of the current layer

Parameters

- **ts_input** (*Optional[TSEvent]*) – TimeSeries of TxM or Tx1 Input signals for this layer
- **duration** (*Optional[float]*) – Duration of the desired evolution, in seconds. If not provided, then either `num_timesteps` or the duration of `ts_input` will determine evolution itme
- **num_timesteps** (*Optional[int]*) – Number of evolution time steps, in units of `.dt`. If not provided, then either `duration` or the duration of `ts_input` will determine evolution time

Return (ndarray, int) spike_raster: Boolean or integer raster containing spike information. T1xM array num_timesteps: Actual number of evolution time steps, in units of `.dt`

property class_name

(str) Class name of `self`

property dt

(float) Simulation time step of this layer

evolve (*ts_input: Optional[rockpool.timeseries.TSContinuous] = None*, *ts_force: Optional[rockpool.timeseries.TSContinuous] = None*, *duration: Optional[float] = None*, *num_timesteps: Optional[int] = None*) → rockpool.timeseries.TimeSeries

evolve() - Evolve the reservoir state

Parameters

- **ts_input** – TSContinuous Input time series
- **ts_force** – TSContinuous Forced time series
- **duration** – float Duration of evolution in seconds
- **num_timesteps** – int Number of time steps to evolve (based on self.dt)

Returns ts_output: TSContinuous Output time series

property input_type

(Type[TimeSeries]) Input *TimeSeries* subclass accepted by this layer.

classmethod load_from_dict (*config: dict*, ***kwargs*) → cls

Generate instance of a *Layer* subclass with parameters loaded from a dictionary

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`

- **config** (*Dict*) – Dictionary containing parameters of a *Layer* subclass
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameters from `config` should be overridden

Return Layer Instance of `cls` with parameters from `config`

classmethod load_from_file (*filename: str*, ***kwargs*) → cls
Generate an instance of a *Layer* subclass, with parameters loaded from a file

Parameters

- **cls** (*Any*) – A *Layer* subclass. This class will be used to reconstruct a layer based on the parameters stored in `filename`
- **filename** (*str*) – Path to the file where parameters are stored
- **kwargs** – Any keyword arguments of the class *__init__* method where the parameter stored in the file should be overridden

Return Layer Instance of `cls` with parameters loaded from `filename`

property noise_std
(float) Noise injected into the state of this layer during evolution

This value represents the standard deviation of a white noise process. When subclassing *Layer*, this value should be correctected by the `dt` attribute

property output_type
(Type[TimeSeries]) Output *TimeSeries* subclass emitted by this layer.

randomize_state()
Randomize the internal state of this layer

Unless overridden, this method randomizes the layer state based on the current state, using a Normal distribution with std. dev. of 20% of the current state values

reset_all()
Reset both the internal clock and the internal state of the layer

reset_state()
Reset the internal state of this layer

Sets *state* attribute to all zeros

reset_time()
Reset the internal clock of this layer to 0

save (*config: dict*, *filename: str*)
Save a set of parameters to a `json` file

Parameters

- **config** (*Dict*) – Dictionary of attributes to be saved
- **filename** (*str*) – Path of file where parameters are stored

save_layer (*filename: str*)
Obtain layer paramters from *to_dict* and save in a `json` file

Parameters **filename** (*str*) – Path of file where parameters are to be stored

property size
(int) Number of units in this layer (N)

property size_in
(int) Number of input channels accepted by this layer (M)

property size_out
(int) Number of output channels produced by this layer (O)

property start_print
(str) Return a string containing the layer subclass name and the layer `name` attribute

property state
(ndarray) Internal state of this layer (N)

property t
(float) The current evolution time of this layer

to_dict() → dict
Convert the layer to a dictionary for saving :return dict:

property w_in
(np.ndarray) [IxN] input weights

property w_out
(np.ndarray) [NxO] output weights

property w_recurrent
(np.ndarray) [NxN] recurrent weights

property weights
(ndarray) Weights encapsulated by this layer (MxN)

- genindex

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

W

X