Neet Documentation

Release 1.0.0

ELIFE

Oct 14, 2019

Contents

Neet is a python package designed to provide an easy-to-use API for creating and evaluating network models. In its current state, Neet supports simulating synchronous Boolean network models, though the API is designed to be model generic. Future work will implement asynchronous update mechanisms and more general network types.

CHAPTER 1

Getting Help

Neet is developed to help people interested in using and analyzing network models to get things done quickly and painlessly. Your feedback is indispensable. Please create an issue if you find a bug, an error in the documentation, or have a feature you'd like to request. Your contribution will make Neet a better tool for everyone.

If you are interested in contributing to Neet, please contact the developers. We'll get you up and running!

Neet Source Repository https://github.com/elife-asu/neet

Neet Issue Tracker https://github.com/elife-asu/neet/issues

CHAPTER 2

Relevant Publications

Daniels, B.C., Kim, H., Moore, D.G., Zhou, S., Smith, H.B., Karas, B., Kauffman, S.A., and Walker, S.I. (2018) "Criticality Distinguishes the Ensemble of Biological Regulatory Networks" *Phys. Rev. Lett.* **121** (13), 138102, doi:10.1103/PhysRevLett.121.138102.

CHAPTER 3

Copyright and Licensing

CHAPTER 4

Contents

4.1 Introduction

Neet is a library for simulating and analyzing dynamical network models. It is written entirely in Python, with minimal external dependencies. It provides a heirarchy of network classes and facilities for analyzing the attractor landscapes, informational structure and *sensitivity* of those network models.

4.1.1 Examples

Neet provides a network classes with methods designed to make common tasks as painless as possible. For example, you can read in a collection of boolean logic equations and immediately probe the dynamics of the network, and compute values such as the `LandscapeMixin.attractors` and the `boolean.SensitivityMixin.average_sensitivity()` of the network

```
>>> from neet.boolean import LogicNetwork
>>> from neet.boolean.examples import MYELOID_LOGIC_EXPRESSIONS
>>> net = LogicNetwork.read_logic(MYELOID_LOGIC_EXPRESSIONS)
>>> net.names
['GATA-2', 'GATA-1', 'FOG-1', 'EKLF', 'Fli-1', 'SCL', 'C/EBPa', 'PU.1', 'cJun',
↪'EgrNab', 'Gfi-1']
>>> net.attractors
array([array([0]), array([62, 38]), array([46]), array([54]),
       array([1216]), array([1116, 1218]), array([896]), array([960])],
      dtype=object)
>>> net.average_sensitivity()
1.0227272727272727
>>> net.network_graph()
<networkx.classes.digraph.DiGraph object at 0x...>
```

See the examples directory of the GitHub repository for Jupyter notebooks which demonstrate some of the Neet's features.

4.1.2 Getting Started

Installation

Dependencies

Neet depends on several packages which will be installed by default when Neet is installed via *pip*:

- six
- numpy
- networkx
- pyinform
- deprecated

However, network visualization is notoriously problematic, and so we have two optional dependencies which are only required if you wish to visualize networks using Neet's builtin capabilities:

- Graphviz
- pygraphviz

True to form, these dependencies are a pain. Graphviz, unfortunately, cannot be installed via pip (see: https://graphviz.gitlab.io/download/ for installation instructions). Once Graphviz has been installed, you can install *pygraphviz* via *pip*.

Via Pip

To install via `pip`, you can run the following

```
$ pip install neet
```

Note that on some systems this will require administrative privileges. If you don't have admin privileges or would prefer to install Neet for your user only, you do so via the `--user` flag:

```
$ pip install --user neet
```

From Source

```
$ git clone https://github.com/elife-asu/neet
$ cd neet
$ python setup.py test
$ pip install .
```

System Support

So far the python wrapper has been tested under `python2.7`, `python3.4` and `python3.5`, and on the following platforms:

Note: We will continue supporting Python 2.7 until January 1, 2020 when PEP 373 states that official support for Python 2.7 will end.

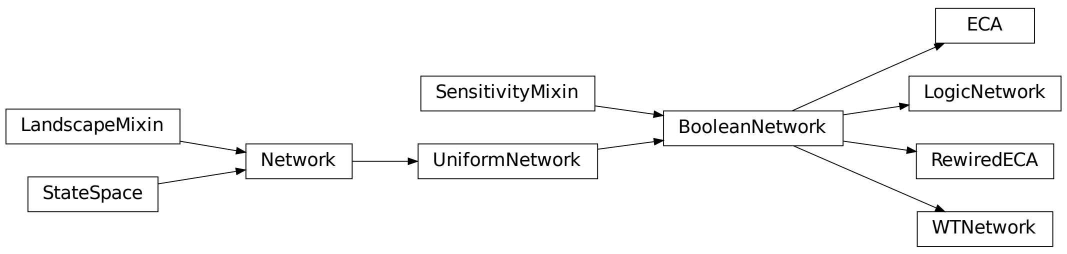
- Debian 8
- Mac OS X 10.11 (El Capitan)
- Windows 10

4.2 Network Classes

Neet provides a collect of pre-defined network types which are common models of complex systems, including

boolean.ECA	ECA represents an elementary cellular automaton rule.
boolean.RewiredECA	RewiredECA represents elementary cellular automaton rule with a rewired topology.
boolean.WTNetwork	WTNetwork represents weight-threshold boolean network.
boolean.LogicNetwork	LogicNetwork represents a network of logic functions.

These concrete network types are leaves in a hierarchy:



All networks in Neet ultimately derive from the *Network* class which provides a uniform interface for all network models. This class provides a basic interface in and of itself, but derives from *StateSpace* and *LandscapeMixin* to provide a wealth of additional features. See *State Spaces* and *Attractor Landscapes* for more information.

4.2.1 Basic Network Attributes

As an example, consider the boolean network *boolean.examples.s_pombe*, which is a gene regulatory network model of the cell cycle of *S. pombe* (fission yeast) [Davidich2008]. All networks have a "shape", namely an array of the number of states each node can take — it's base.

```
>>> s_pombe.shape
[2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Along with this, comes the ability to ask how many nodes the network has:

```
>>> s_pombe.size
9
```

In general, Neet's networks need not be uniform; each state can have a different base. However, all of the networks currently implemented are Boolean, meaning that every node in the network has a binary.

In addition to specifying the base of the nodes of the network, each node can be given a name.

```
>>> s_pombe.names
['SK', 'Cdc2_Cdc13', 'Ste9', 'Rum1', 'Slp1', 'Cdc2_Cdc13_active', 'Wee1_Mik1', 'Cdc25
↪', 'PP']
```

Further, on the whole you can associate an arbitrary dictionary of metadata data, for example citation information.

```
>>> s_pombe.metadata['citation']
'M. I. Davidich and S. Bornholdt, "Boolean network model predicts cell cycle sequence␣
↪of fission yeast," PLoS One, vol. 3, no. 2, p. e1672, Feb. 2008.doi:10.1371/journal.
↪pone.0001672'
```

4.2.2 Dynamic State Update

Beyond data such as these, concrete classes specify information necessary for describing the dynamics of the network's state. Unlike most dynamical network packages, Neet's networks do not store the state of the network internally. Instead, the API provides methods for operating on state external to the network. In particular, `Network.update()` which updates a state of the list or `numpy.ndarray` **in place**.

```
>>> state = [0, 1, 1, 0, 1, 0, 0, 1, 0]
>>> s_pombe.update(state)
[0, 0, 0, 0, 0, 0, 0, 1, 1]
>>> state
[0, 0, 0, 0, 0, 0, 0, 1, 1]
```

This single function allows Neet to implement a number of common analyses such as *landscape*, *information* and *sensitivity* analyses.

4.2.3 Graph Structure

As dynamical networks, all `Network` instances have a directed graph structure. Neet provides a minimal interface for exploring this structure. At it's basic, you can probe which nodes are connected by an edge:

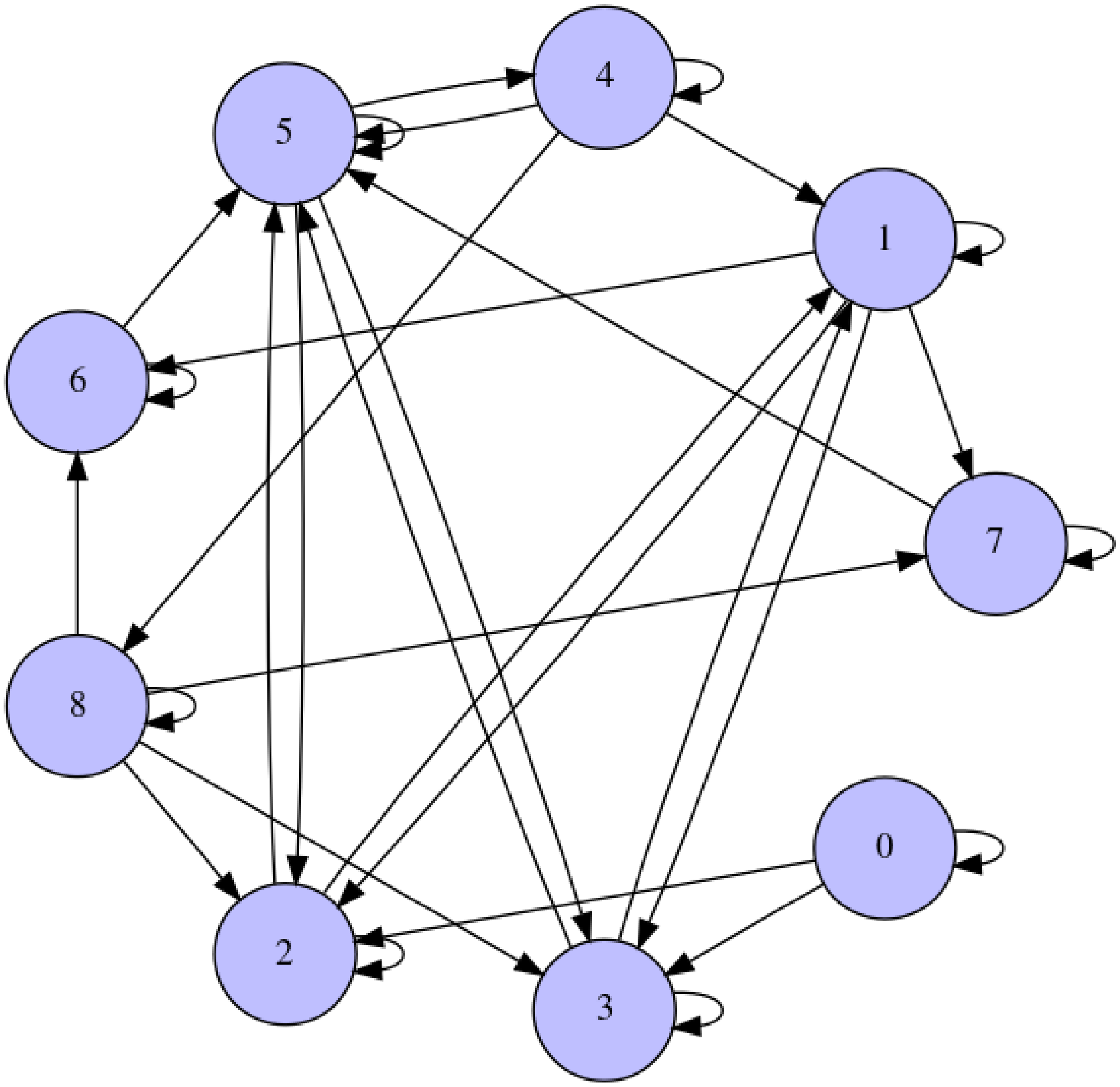
```
# source nodes of edges incoming to node 6
>>> s_pombe.neighbors(6, direction='in') == {1, 6, 8}
True

# target nodes of edges outgoing from 6
>>> s_pombe.neighbors(6, direction='out') == {5, 6}
True

# all nodes connected to node 6
>>> s_pombe.neighbors(6, direction='both') == {1, 5, 6, 8}
True

# all nodes connected to node 6
>>> s_pombe.neighbors(6, direction='both') == {1, 5, 6, 8}
True
```

Of course, this will only get you so far. Luckily, the NetworkX package provides a whole host of graph-theoretic analyses. To take advantage of that fact, and not avoid Neet reinventing the wheel, you can export your Neet network as a `networkx.DiGraph`.

```
>>> import networkx as nx
>>> g = s_pombe.network_graph()
>>> nx.shortest_path(g, 1, 5)
[1, 2, 5]
>>> g = s_pombe.network_graph(labels='names')  # default labels='indices'
>>> nx.shortest_path(g, 'Cdc2_Cdc13', 'Cdc2_Cdc13_active')
['Cdc2_Cdc13', 'Ste9', 'Cdc2_Cdc13_active']
```

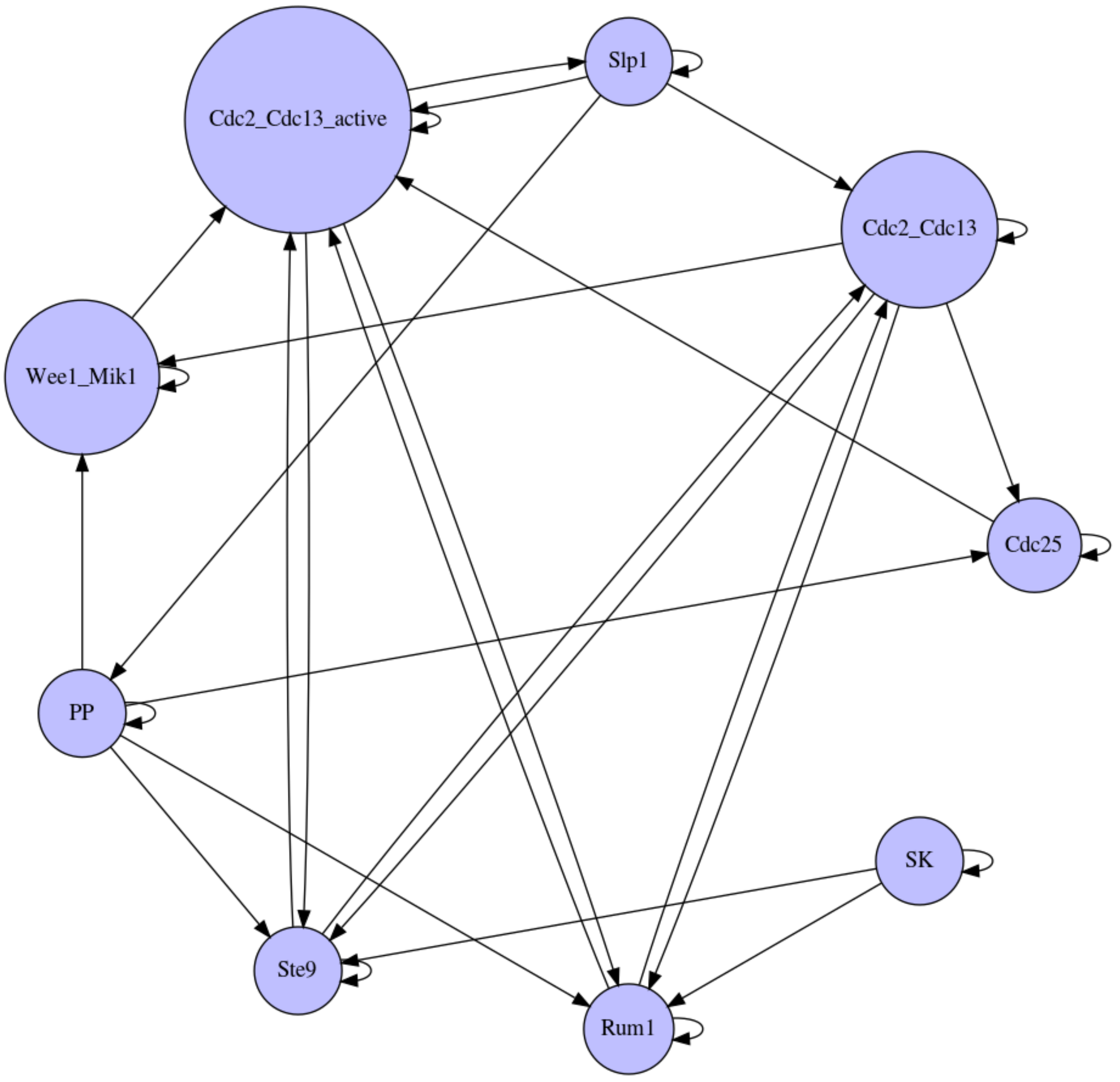
You can draw the graphs, with the nodes labeled by either the node index

```
>>> s_pombe.draw_network_graph({'labels': 'indices'}, {
...     'path': 'source/static/s_pombe_indices.png',
...     'display_image': False
... })
```



or labeled by the node name:

```
>>> s_pombe.draw_network_graph({'labels': 'names'}, {
...     'path': 'source/static/s_pombe_names.png',
...     'display_image': False
... })
```



Note: For the drawing functionality, you will need to install the optional dependencies: Graphviz and pygraphviz. See Getting Started.

4.3 State Spaces

Network derives from *StateSpace* which endows it with structural information about the state space of the network, and provides a number of vital methods.

4.3.1 Attributes

First and foremost, *StateSpace* provides (readonly) attributes for assessing gross properties of the state space, namely *StateSpace.size*, *StateSpace.shape* and *StateSpace.volume*.

```
>>> s_pombe.size  # number of dimension (nodes)
9
>>> s_pombe.shape  # the number of states by dimension (states per node)
[2, 2, 2, 2, 2, 2, 2, 2, 2]
>>> s_pombe.volume  # total number of states of the network
512
```

4.3.2 States in the Space

As a *StateSpace*, you can determining whether or not an array represents a valid state of the network. This is accomplished using the `in` keyword.

```
>>> 0 in s_pombe
False
>>> [0]*9 in s_pombe
True
>>> numpy.zeros(9, dtype=int) in s_pombe
True
>>> [2, 0, 0, 0, 0, 0, 0, 0, 0] in s_pombe  # the nodes are binary
False
```

Of course, after asking whether a state is valid, the next thing you might want to do is iterate over the states.

```
>>> for state in s_pombe:
...     print(state)
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0]
...
[0, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Since the networks are iterable, you can treat them like any other kind of sequence.

```
>>> list(s_pombe)
[[0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0], ...]
>>> list(map(lambda s: s[0], s_pombe))
[0, 1, 0, 1, ...]
>>> list(filter(lambda s: s[0] ^ s[1] == 1, s_pombe))
[[1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0], ...]
```

4.3.3 State Encoding and Decoding

For particularly large networks, storing a list of states it's states can use a lot of memory. What's more, it is often useful to be able to index an array or key a dictionary based by a state of the network, e.g. when efficiently computing the attractors of the network. A simple solution to this problem is to encode the state as an integer. *StateSpace* provides this functionality via the *StateSpace.encode()* and *StateSpace.decode()* methods.

Encoding States

```
>>> s_pombe.encode([0, 1, 0, 1, 0, 1, 0, 1, 0])
170
>>> s_pombe.encode(numpy.ones(9)) == s_pombe.volume - 1
True
>>> s_pombe.encode('apples')
Traceback (most recent call last):
...
ValueError: state is not in state space
```

Decoding States

```
>>> s_pombe.decode(170)
[0, 1, 0, 1, 0, 1, 0, 1, 0]
>>> s_pombe.decode(511)
[1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> s_pombe.decode(512)
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> s_pombe.decode(-1)
[1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Notice that decoding states does not raise an error when the state encoding is invalid. Instead, the codes wrap around so that any integer can be decoded. This was a decision made more for the sake of performance than anything. Just be mindful of it.

By and large, the `StateSpace.encode()` and `StateSpace.decode()` methods are inverses:

```
>>> s_pombe.encode(s_pombe.decode(170))
170
>>> s_pombe.decode(s_pombe.encode([0, 0, 1, 0, 0, 1, 0, 0, 1]))
[0, 0, 1, 0, 0, 1, 0, 0, 1]
```

4.3.4 Encoding Scheme

There are a number of ways of encoding a sequence of integers as an integer. We've chosen the one we did so that the encoded value of the state is consistent with the order the states are produced upon iteration.

```
>>> states = list(s_pombe)
>>> states[5] == s_pombe.decode(5)
True
>>> numpy.all([i == s_pombe.encode(s) for i, s in enumerate(s_pombe)])
True
>>> numpy.all([s_pombe.decode(i) == s for i, s in enumerate(s_pombe)])
True
```

This makes implementing the algorithms associated with *landscape dynamics* and *sensitivity analyses* much simpler and as light on memory as possible.

4.4 Attractor Landscapes

The most common use of dynamical network models is the analysis of their attractor landscape. In many cases, the attractors are associated with some form of functionally important network state, e.g. a cell type in a gene regulatory network. Neet provides standard landscape analysis methods via the `LandscapeMixin` from which `Network` derives.

4.4.1 State Transitions

The starting point for all of these analyses are the state transitions: where does each state of the network go upon update?

```
>>> s_pombe.transitions
array([  2,   2, 130, 130,   4,   0, 128, 128,   8,   0, 128, 128,  12,
         0, 128, 128, 256, 256, 384, 384, 260, 256, 384, 384, 264, 256,
       ...
       208, 208, 336, 336, 464, 464, 340, 336, 464, 464, 344, 336, 464,
       464, 348, 336, 464, 464])
```

Each element of the resulting array is the state to which the index transitions, e.g. $0 \mapsto 2$, $2 \mapsto 130$, etc. The indices and values are, of course, *encoded* states. You can always decode them:

```
>>> for x, y in enumerate(s_pombe.transitions):
...     print(s_pombe.decode(x), '→', s_pombe.decode(y))
[0, 0, 0, 0, 0, 0, 0, 0, 0] → [0, 1, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0] → [0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0] → [0, 1, 0, 0, 0, 0, 0, 1, 0]
...
[1, 1, 1, 1, 1, 1, 1, 1, 1] → [0, 0, 0, 0, 1, 0, 1, 1, 1]
```

Given state transitions, the next question you might ask is how to compute sequences of state transtions — a trajectory — by applying the network update scheme recursively?

```
>>> s_pombe.trajectory([0, 0, 0, 0, 0, 0, 0, 0, 0], timesteps=2)
[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 1,␣
↪0]]
>>> s_pombe.trajectory([0, 0, 0, 0, 0, 0, 0, 0, 0], timesteps=2, encode=True)
[0, 2, 130]
```

Notice that if you request a trajectory with t time steps, the resulting trajectory will have $t + 1$ elements in it; the first element is the initial state. If you want the trajectory for *every* state of the network, you can use the timeseries method.

```
>>> series = s_pombe.timeseries(2)
>>> series
array([[[0, 0, 0],
        [1, 0, 0],
        [0, 0, 0],
        ...,
        [1, 0, 0],
        [0, 0, 0],
        [1, 0, 0]],

       ...

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [1, 1, 1],
        [1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
        [1, 1, 1]]])
>>> series.shape
(9, 512, 3)
>>> series[:, 0, :].transpose()
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 1, 0]])
```

The resulting 3-D array is indexed by the nodes, state and timestep; in that order. For a more wholistic description of the state transitions, you can construct a landscape graph.

```
>>> import networkx as nx
>>> g = s_pombe.landscape_graph()
>>> len(g)
512
>>> nx.shortest_path(g, 0, 130)
[0, 2, 130]
```

The landscape graph, much like the network topology, can be drawn if you've installed pygraphviz. See Getting Started.

4.4.2 Attractors and Basins

With the state transitions under our belt, we can start computing landscape features such as the attractors.

```
>>> s_pombe.attractors
array([array([76]), array([4]), array([8]), array([12]),
       array([144, 110, 384]), array([68]), array([72]), array([132]),
       array([136]), array([140]), array([196]), array([200]),
       array([204])], dtype=object)
```

Each element of the resulting array is an array of states in a fixed-point attractor or limit cycle. Beyond this, you can determine which of the attractor's basin each state is in.

```
>>> s_pombe.basins
array([ 0,  0,  0,  0,  1,  0,  0,  0,  2,  0,  0,  0,  3,  0,  0,  0,  0,
        0,  4,  4,  0,  0,  4,  4,  0,  0,  4,  4,  0,  0,  4,  4,  4,  4,
        ...
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0])
>>> s_pombe.basins[18]
4
```

That is, state 18 is in basin 4, and so is fated to land in the cycle $\{144, 110, 384\}$.

The *LandscapeMixin* provides a whole host of other properties, so check out the API Reference for the full list.

4.4.3 Landscape Data

A key feature of the *LandscapeMixin* is that it tries to compute as much as it can, as efficiently as it can. For example, when the attractors are computed, the basins of all of the states, the recurrence time, etc. . . can all be computed at the same time. These values are

1. **Computed lazily, but preemptively** when you first request any of the associated property.

2. **Cached** in a `LandscapeData` object stored in the `LandscapeMixin`.

This means, that the attractors are computed when you request them. A second request will simply use the cached values. Similarly, you get a cached value for the basins once you've accessed the attractors. The following only computes the attractors once, and the basins are computed at that call:

```
>>> s_pombe.attractors  # may take a moment
array([...], dtype=object)
>>> s_pombe.attractors  # almost instantaneous
array([...], dtype=object)
>>> s_pombe.basins  # almost instantaneous; computed on first call to attractors.
array([...])
```

The order you access the properties in does not matter, so don't worry about that.

There may be cases when you want to

1. Compute some landscape features of a network
2. Modify the network in some way
3. Compute landscape features on the new network
4. Compare the results

Because you've modifed the network, you will need to reset the cached landscape data. Since you are going to be comparing features before and after, you need to extract the data before you do that. This is where `LandscapeMixin.landscape()`, `LandscapeMixin.expound()` and `LandscapeMixin.landscape_data` come into play.

```
import numpy
from neet.boolean.examples import s_pombe

# Compute all of the landscape properties
s_pombe.expound()
# Get the data out
before = s_pombe.landscape_data

# Modify the network
s_pombe.thresholds = numpy.zeros(s_pombe.size)
# Reset the landscape (notice the method chaining...)
s_pombe.landscape().expound()
# Get the new data
after = s_pombe.landscape_data

# Compare `before` and `after` as you so choose
```

The result of `LandscapeMixin.landscape_data` is a `LandscapeData` object which has all of the landscape features cached (provided they've been computed):

```
>>> s_pombe.attractors
array([...], dtype=object)
>>> s_pombe.landscape_data
<neet.landscape.LandscapeData object at 0x...>
>>> s_pombe.landscape_data.attractors
array([...], dtype=object)
```

4.5 Information Analysis

Out of the box, Neet provides facilities for computing a few common information-theoretic quantities from networks. All of these methods rely on constructing time series, from which a collection of probabilities distributions are built. The *Information* class provides a simple mechanism for automating this process, and caching results for relatively efficient computation.

4.5.1 Initialization

Constructing an instance of *Information*, you simply provide a network, a history length (used to compute measures such as active information or transfer entropy), and the length of time series to compute.

```
>>> Information(s_pombe, k=5, timesteps=20)
<neet.information.Information object at 0x...>
```

At initialization, a time series is computed based on the parameters provided. This is cached and used whenever you request an information measure.

Of course, you can override the parameters after initialization, and the time series will be recomputed.

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.net = s_cerevisiae
>>> arch.k = 2
>>> arch.timesteps = 100
```

4.5.2 Information Measures

Once you have an *Information* instance, you can request an informormation measure. This will compute and cache the value.

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.active_information()  # computed and cached
array([0.        , 0.4083436 , 0.62956679, 0.62956679, 0.37915718,
       0.40046165, 0.67019615, 0.67019615, 0.39189127])
>>> arch.active_information()  # cached value is returned
array([0.        , 0.4083436 , 0.62956679, 0.62956679, 0.37915718,
       0.40046165, 0.67019615, 0.67019615, 0.39189127])
```

Each information measure is only computed and cached when you request it. In the event that you change some aspect of the information architecture, e.g. the network, the cache of information measures is also cleared.

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.active_information()
array([0.        , 0.4083436 , 0.62956679, 0.62956679, 0.37915718,
       0.40046165, 0.67019615, 0.67019615, 0.39189127])
>>> arch.net = s_cerevisiae
>>> arch.active_information()
array([0.        , 0.35677758, 0.410884  , 0.44191249, 0.54392362,
       0.42523414, 0.35820287, 0.13355861, 0.42823889, 0.22613507,
       0.28059538])
```

4.6 Sensitivity Analysis

Neet provides an API for computing various measures of sensitivity on Networks via the `SensitivityMixin`. Sensitivity, in its simplest form, is a measure of how small perturbations of the network's state change under the dynamics. In the sensitivity parlance, a network is called, *sub-critical*, *critical*, or *chaotic* if the perturbation tends to shrink, stay the same, or grow over time.

Note: As of the v1.0.0 release, only the `neet.boolean` module provides implementations of the sensitivity interface. A subsequent release will generalize this mixin to support a wider range of network models.

4.6.1 Boolean Sensitivity

The standard definition of sensitivity at a given state of a Boolean network is defined in terms of the Hamming distance:

$$D_H(x, y) = \sum_i x_i \oplus y_i.$$

That is, the number of bits differing between two binary states, x and y. A Hamming neighbor of a state x is a state that differs from it by exactly 1 bit. We can write $x \oplus e_i$ to represent the Hamming neighbor of x which differs in the i-th bit. The sensitivity of the state x is then defined as

$$s_f(x) = \frac{1}{N}\sum_{i=1}^{N} D_H(f(x), f(x \oplus e_i))$$

where f is the network's update function, and N is the number of nodes in the network.

Neet makes computing sensitivity at a given network state as straightforward as possible:

```
>>> s_pombe.sensitivity([0, 0, 0, 0, 0, 0, 0, 0 ,0])
1.5555555555555556
```

More often than not, though, you'll want to compute the average of the sensitivity over all of the states of the network. That is

$$s_f = \frac{1}{2^N}\sum_x s_f(x).$$

In Neet, just ask for it

```
>>> s_pombe.average_sensitivity()
0.9513888888888888
```

For a full range of sensitivity-related features offered by Neet, see the API References.

4.7 Network Randomization

The previous release of Neet v0.1.0 included the `neet.boolean.randomnet` module which provided mechanisms for randomizing network models (Boolean networks, more specificially). However, the maintainers are not quite satisified with the quality and scope of that module. Rather than postpone the v1.0 release any longer, we have decided to withhold that module for the time begin.

The current plans are to release a totally redesigned module in the future, possibly with v2.0.

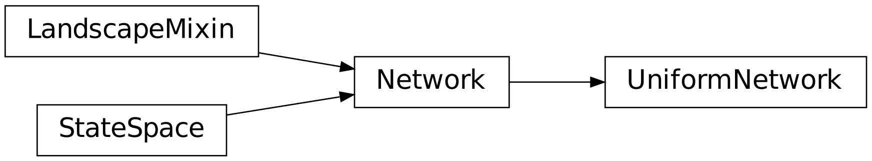
If this is a feature that you desperately need, please feel free to email the developers at emergence@asu.edu or comment on the relevant issue on GitHub.

4.8 API Reference

4.8.1 Network Classes

The `neet` module provides the following abstract network classes from which all concrete Neet networks inherit:

`Network`	The Network class is the core base class for all Neet networks.
`UniformNetwork`	The UnformNetwork class represents a network in which every node has the same number of discrete states.



These classes provide an abstract interface which algorithms can leverage for generic implementation of various network-theoretic analyses.

Network

class neet.**Network**(*shape*, *names=None*, *metadata=None*)

The Network class is the core base class for all Neet networks. It provides an interface for describing network state updating and simple graph-theoretic analyses.

`names`	Get or set the names of the nodes of the network.
`metadata`	Any metadata associated with the network.
`_unsafe_update`	Unsafely update the state of a network in place.
`update`	Update the state of a network in place.
`neighbors_in`	Get a set of all incoming neighbors of the node at `index`.
`neighbors_out`	Get a set of all outgoing neighbors of the node at `index`.
`neighbors`	Get a set of the neighbors of the node at `index`.
`network_graph`	The graph of the network as a `networkx.DiGraph`.
`draw_network_graph`	Draw network's networkx graph using PyGraphviz.

Network is an *abstract* class, meaning it cannot be instantiated, and inherits from `neet.LandscapeMixin` and `neet.StateSpace`. Initialization of the Network requires, at a minimum, a specification of the shape of the network's state space, and optionally allows the user to specify a list of names for the nodes of the network and a metadata dictionary for the network as a whole (e.g. citation information).

Any concrete deriving class must overload the following methods:

- *_unsafe_update()*
- *neighbors_in()*
- *neighbors_out()*

Parameters

- **shape** (*list*) – the base of each node of the network
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

metadata
Any metadata associated with the network.

names
Get or set the names of the nodes of the network.

Raises

- **TypeError** – if the assigned value is not convertable to a list
- **ValueError** – if the length fo the assigned values does not match the networks's size

_unsafe_update(*state*, *index*, *pin*, *values*, **args*, ***kwargs*)
Unsafely update the state of a network in place.

This function accepts three optional arguments by default:

- `index` - update only the specified node (by index)
- `pin` - do not update the state of any node in a list
- `values` - set the state of some subset of nodes to specified values

Note: As an abstract method, every concrete class derving from Network must overload this method. The overload **should not** perform no ensurance checks on the arguments to maximize performance, as those check are performed in the *update()* method. Further, it is assumed that this method *modifies* the `state` argument in-place and no others.

Parameters

- **state** (*list*, *numpy.ndarray*) – the state of the network to update
- **index** (*int or None*) – the index to update
- **pin** (*list*, *numpy.ndarray or None*) – which nodes to pin to their current state
- **values** (*dict or None*) – a dictionary mapping nodes to a state to which to reset the node to

Returns the updated state

neighbors_in(*index*, **args*, ***kwargs*)
Get a set of all incoming neighbors of the node at `index`.

All concrete network classes must overload this method.

Parameters **index** (*int*) – the index of the node target node

Returns a set of incoming neighbor indices

neighbors_out (*index*, **args*, ***kwargs*)
Get a set of all outgoing neighbors of the node at `index`.

All concrete network classes must overload this method.

Parameters **index** (*int*) – the index of the node source node

Returns a set of outgoing neighbor indices

update (*state*, *index=None*, *pin=None*, *values=None*, **args*, ***kwargs*)
Update the state of a network in place.

This function accepts three optional arguments by default:

- `index` - update only the specified node (by index)
- `pin` - do not update the state of any node in a list
- `values` - set the state of some subset of nodes to specified values

Examples

Updates States In-Place:

```
>>> rule = ECA(30, size=5)
>>> state = [0, 0, 1, 0, 0]
>>> rule.update(state)
[0, 1, 1, 1, 0]
>>> state
[0, 1, 1, 1, 0]
```

Updating A Single Node:

```
>>> rule = ECA(30, size=5)
>>> rule.update([0, 0, 1, 0, 0])
[0, 1, 1, 1, 0]
>>> rule.update([0, 0, 1, 0, 0], index=1)
[0, 1, 1, 0, 0]
```

Pinning States:

```
>>> rule = ECA(30, size=5)
>>> rule.update([0, 0, 1, 0, 0])
[0, 1, 1, 1, 0]
>>> rule.update([0, 0, 1, 0, 0], pin=[1])
[0, 0, 1, 1, 0]
```

Overriding States:

```
>>> rule = ECA(30, size=5)
>>> rule.update([0, 0, 1, 0, 0])
[0, 1, 1, 1, 0]
>>> rule.update([0, 0, 1, 0, 0], values={0: 1, 2: 0})
[1, 1, 0, 1, 0]
```

This function ensures that:

1. If `index` is provided, then neither `pin` nor `values` is provided.

2. If `pin` and `values` are both provided, then they do not affect the same nodes.
3. If `values` is provided, then the overriding states specified in it are consistent with the state space of the network.

Note: Typically, this method should not be overloaded unless the particular deriving class makes use of the `args` or `kwargs` arguments. In that case, it should first ensure that those arguments are well-behaved, and and the delegate subsequent checks and the call to *`_unsafe_update()`* to a call to this *`neet.Network.update()`*.

Parameters

- **`state`** (*`list or numpy.ndarray`*) – the state of the network to update
- **`index`** (*`int or None`*) – the index to update
- **`pin`** (*`list, numpy.ndarray or None`*) – which nodes to pin to their current state
- **`values`** (*`dict or None`*) – a dictionary mapping nodes to a state to which to reset the node to

Returns the updated state

`neighbors`(*index*, *direction='both'*, **args*, ***kwargs*)

Get a set of the neighbors of the node at `index`. Optionally, specify the directionality of the neighboring edges, e.g. `'in'`, `'out'` or `'both'`.

Examples

All Neighbors:

```
>>> s_pombe.neighbors(7)
{1, 5, 7, 8}
```

Incoming Neighbors:

```
>>> s_pombe.neighbors(7, direction='in')
{8, 1, 7}
```

Outgoing Neighbors:

```
>>> s_pombe.neighbors(7, direction='out')
{5, 7}
```

Parameters

- **`index`** (*`int`*) – the index of the node
- **`direction`** (*`str`*) – the directionality of the neighboring edges

Returns a set of neighboring node indices, respecting `direction`.

`network_graph`(*labels='indices'*, ***kwargs*)

The graph of the network as a `networkx.DiGraph`.

This method should only be overloaded by derived classes if additional metadata is to be added to the graph by default.

Examples

```
>>> s_pombe.network_graph()
<networkx.classes.digraph.DiGraph object at 0x...>
```

Parameters

- **labels** – label to be applied to graph nodes (either `'indices'` or `'names'`)
- **kwargs** – kwargs to pass to the `networkx.DiGraph` constructor

Returns a `networkx.DiGraph` object

draw_network_graph(*graphkwargs={}*, *pygraphkwargs={}*)
Draw network's networkx graph using PyGraphviz.

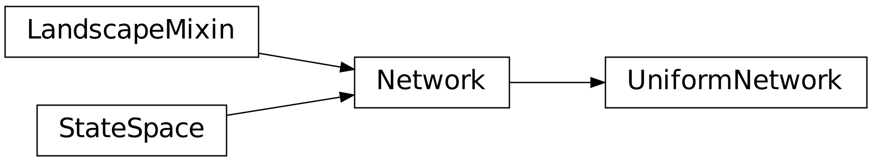
Note: This method requires Graphviz and pygraphviz. The former requires manual installation (see https://graphviz.gitlab.io/download/), while the latter can be installed via `pip`.

Parameters

- **graphkwargs** – kwargs to pass to *`network_graph()`*
- **pygraphkwargs** – kwargs to pass to *`neet.draw.view_pygraphviz()`*

UniformNetwork

class neet.**UniformNetwork**(*size*, *base*, *names=None*, *metadata=None*)
The UnformNetwork class represents a network in which every node has the same number of discrete states. This allows for more efficient default implementations of several methods. If your particular concrete network type meets this condition, then you should derive from UniformNetwork rather than Network.



In addition to the methods provided by *`Network`*, UniformNetwork also provides the following attribute:

`base`	Get the number of states each node can take.

UniformNetwork derives from *`Network`*, but is still *abstract*, meaning it cannot be instantiated. Initialization of the *`UniformNetwork`* requires, at a minimum, the number of nodes in the network (`size`) and the number of states the nodes can take (`base`). As with *`Network`*, the user can optionally specify a list of names for the nodes of the network and a metadata dictionary for the network as a whole (e.g. citation information).

Any concrete deriving class must overload the following methods:

- `_unsafe_update()`
- `neighbors_in()`
- `neighbors_out()`

Parameters

- **size** (*int*) – the number of nodes in the network
- **base** (*int*) – the number of states each node can take
- **names** (*seq*) – an interable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

base

Get the number of states each node can take.

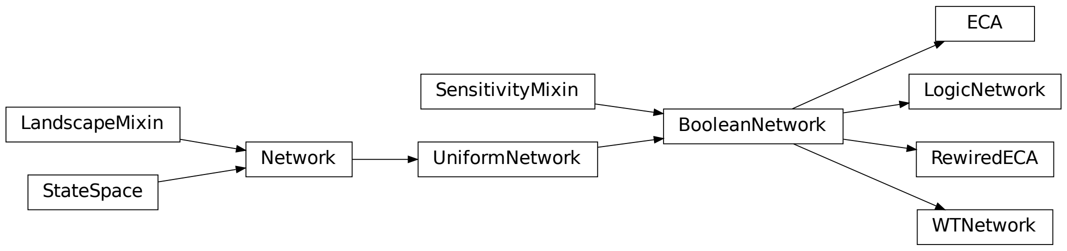
Examples

```
>>> ECA(30, size=5).base
2
```

Returns the base of nodes of the network

4.8.2 Boolean Networks

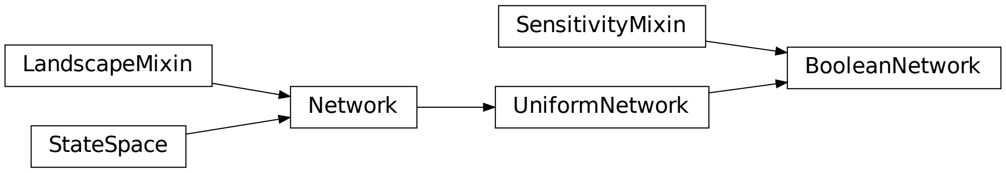
BooleanNetwork	The BooleanNetwork class is a base class for all of Neet's Boolean networks.
ECA	ECA represents an elementary cellular automaton rule.
RewiredECA	RewiredECA represents elementary cellular automaton rule with a rewired topology.
WTNetwork	WTNetwork represents weight-threshold boolean network.
LogicNetwork	LogicNetwork represents a network of logic functions.



BooleanNetwork

class neet.boolean.**BooleanNetwork**(*size*, *names=None*, *metadata=None*)

The BooleanNetwork class is a base class for all of Neet's Boolean networks. The BooleanNetwork class inherits from both *neet.UniformNetwork* and *neet.boolean.SensitivityMixin*, and specializes the inherited *neet.StateSpace* methods to exploit the Boolean structure.



In addition to all of its inherited methods, BooleanNetwork also exposes the following methods:

subspace	Generate all states in a given subspace.
distance	Compute the Hamming distance between two states.
hamming_neighbors	Get all states that one unit of Hamming distance from a given state.

BooleanNetwork is an *abstract* class, meaning it cannot be instantiated. Initialization of a BooleaNetwork requires, at a minimum, the number of nodes in the network. As with all classes that derive from *neet.Network*, the user may optionally provide a list of names for the nodes of the network and a metadata dictionary for the network as a whole (e.g. citation information).

Parameters

- **size** (*int*) – number of nodes in the network
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

subspace(*indices*, *state=None*)

Generate all states in a given subspace. This method varies each node specified by the `indicies` array independently. The optional `state` parameter specifies the state of the non-varying states of the network. If `state` is not provided, all nodes not in `indicies` will have state `0`.

Examples

```
>>> s_pombe.subspace([0])
<generator object BooleanNetwork.subspace at 0x...>
>>> list(s_pombe.subspace([0]))
[[0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0]]
>>> list(s_pombe.subspace([0, 3]))
[[0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0,␣
↪0, 0, 0], [1, 0, 0, 1, 0, 0, 0, 0, 0]]
```

```
>>> s_pombe.subspace([0], state=[0, 1, 0, 1, 0, 1, 0, 1, 0])
<generator object BooleanNetwork.subspace at 0x...>
>>> list(s_pombe.subspace([0], state=[0, 1, 0, 1, 0, 1, 0, 1, 0]))
[[0, 1, 0, 1, 0, 1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 1, 0]]
>>> list(s_pombe.subspace([0, 3], state=[0, 1, 0, 1, 0, 1, 0, 1, 0]))
[[0, 1, 0, 1, 0, 1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 1, 0], [0, 1, 0, 0, 0, 1,␣
↪0, 1, 0], [1, 1, 0, 0, 0, 1, 0, 1, 0]]
```

Parameters

- **indicies** (*list, numpy.ndarray, iterable*) – the indicies to vary in the subspace
- **state** (*list, numpy.ndarray*) – a state which specifes the state of the non-varying nodes

Yield the states of the subspace

distance (*a, b*)

Compute the Hamming distance between two states.

Examples

```
>>> s_pombe.distance([0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 1, 1, 0, 1, 0, 0])
4
>>> s_pombe.distance([0, 1, 0, 1, 1, 0, 1, 0, 0], [0, 1, 0, 1, 1, 0, 1, 0, 0])
0
```

Parameters

- **a** (*list, numpy.ndarray*) – the first state
- **b** (*list, numpy.ndarray*) – the second state

Returns the Hamming distance between the states

Raises **ValueError** – if either state is not in the network's state space

hamming_neighbors (*state*)

Get all states that one unit of Hamming distance from a given state.

Examples

```
>>> s_pombe.hamming_neighbors([0, 0, 0, 0, 0, 0, 0, 0, 0])
[[1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0,␣
↪0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0, 0], [0, 0,␣
↪0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1,␣
↪0], [0, 0, 0, 0, 0, 0, 0, 0, 1]]
>>> s_pombe.hamming_neighbors([0, 1, 1, 0, 1, 0, 1, 0, 0])
[[1, 1, 1, 0, 1, 0, 1, 0, 0], [0, 0, 1, 0, 1, 0, 1, 0, 0], [0, 1, 0, 0, 1, 0,␣
↪1, 0, 0], [0, 1, 1, 1, 1, 0, 1, 0, 0], [0, 1, 1, 0, 0, 0, 1, 0, 0], [0, 1,␣
↪1, 0, 1, 1, 1, 0, 0], [0, 1, 1, 0, 1, 0, 0, 0, 0], [0, 1, 1, 0, 1, 0, 1, 1,␣
↪0], [0, 1, 1, 0, 1, 0, 1, 0, 1]]
```

Parameters **state** (*list, numpy.ndarray*) – the state whose neighbors are desired

Returns a list of neighbors of the given state

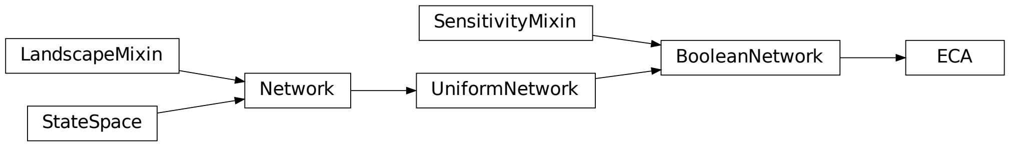
Raises `ValueError` – if the state is not in the network's state space

Elementary Cellular Automata

The *`neet.boolean.ECA`* class describes an Elementary Cellular Automaton with an arbitrary rule.

class `neet.boolean.`**`ECA`**(*code*, *size*, *boundary=None*, *names=None*, *metadata=None*)

ECA represents an elementary cellular automaton rule. Each ECA contains an 8-bit integral member variable `code` representing the Wolfram code for the ECA rule and a set of boundary conditions which is either `None`, signifying periodic boundary conditions, or a pair of cell states signifying fixed, open boundary conditions. As with all *`neet.Network`* classes, the names of the nodes and network-wide metadata can be provided.



In addition to all inherited methods, ECA exposes the following properites:

`code`	The Wolfram code of the elementary cellular automaton.
`boundary`	The boundary conditions of the elemenary cellular automaton.

Parameters

- **`code`** (*`int`*) – the Wolfram code for the ECA
- **`size`** (*`int`*) – the size of the ECA's lattice
- **`boundary`** (*`tuple or None`*) – the boundary conditions for the CA
- **`names`** (*seq*) – an iterable object of the names of the nodes in the network
- **`metadata`** (*`dict`*) – metadata dictionary for the network

Raises

- `ValueError` – if `code` is not in $\{0, 1, \ldots, 255\}$
- `ValueError` – if `boundary` is a neither `None` nor a pair of binary states

`code`

The Wolfram code of the elementary cellular automaton.

Examples

```
>>> eca = ECA(30, size=5)
>>> eca.code
30
>>> eca.code = 45
>>> eca.code
45
>>> eca.code = 256
Traceback (most recent call last):
    ...
ValueError: invalid ECA code
```

Type int

Raises **`ValueError`** – if `code` is not in $\{0, 1, \ldots, 255\}$

`boundary`
The boundary conditions of the elemenary cellular automaton.

Examples

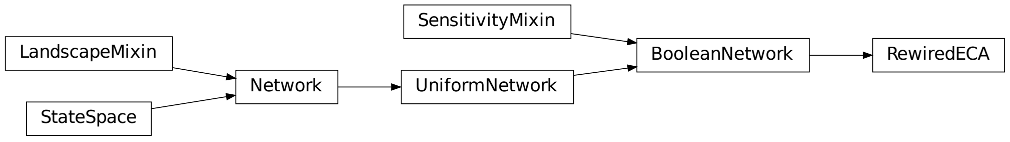
```
>>> eca = ECA(30, size=5)
>>> eca.boundary
>>> eca.boundary = (0, 1)
>>> eca.boundary
(0, 1)
>>> eca.boundary = None
>>> eca.boundary
>>> eca.boundary = [0, 1]
Traceback (most recent call last):
    ...
TypeError: ECA boundary are neither None nor a tuple
```

Type tuple, None

Raises **`ValueError`** – if `boundary` is a neither `None` nor a pair of binary states

Rewired Elementary Cellular Automata

class `neet.boolean.`**`RewiredECA`**(*code*, *boundary=None*, *size=None*, *wiring=None*, *names=None*, *metadata=None*)
RewiredECA represents elementary cellular automaton rule with a rewired topology. That is, RewiredECA is a variant of an *`neet.boolean.ECA`* wherein the neighbors of a given cell can be specified by the user. This allows one to study, for example, the role of topology in the dynamics of a network. Every *`neet.boolean.ECA`* can be represented as a RewiredECA with standard wiring, but all RewiredECA are *fixed sized* networks. For this reason, RewiredECA **does not** derive from *`neet.boolean.ECA`*.



RewiredECA instances can be instantiated by providing an ECA rule `code`, and either the number of nodes in the network (`size`) or a `wiring` matrix which specifies how the nodes are wired. Optionally, the user can specify boundary conditions as in *`neet.boolean.ECA`*. As with all *`neet.Network`* classes, the names of the nodes and network-wide metadata can be provided.

In addition to all inherited methods, RewiredECA exposes the following properites

`code`	The Wolfram code of the elementary cellular automaton
`boundary`	The boundary conditions of the elemenary cellular automaton
`wiring`	The wiring matrix for the rule.

Examples

If `wiring` is not provided, the network is wired as a standard *`neet.boolean.ECA`*.

```
>>> reca = RewiredECA(30, size=5)
>>> reca.code
30
>>> reca.size
5
>>> reca.wiring
array([[-1,  0,  1,  2,  3],
       [ 0,  1,  2,  3,  4],
       [ 1,  2,  3,  4,  5]])
```

Wiring matrices are $3imesN$ matrices where each column is a node of the network, and the rows represent the left-, middle- and right-input for the nodes. The number of nodes will be inferred from the width of the matrix. For example:

```
>>> reca = RewiredECA(30, wiring=[[0,1,2],[-1,0,0],[2,3,1]])
>>> reca.code
30
>>> reca.size
3
>>> reca.wiring
array([[ 0,  1,  2],
       [-1,  0,  0],
       [ 2,  3,  1]])
```

Here the 0, -1 and 2 as left, middle and right input. Note that `-1` represents the left-boundary condition of the RewiredECA. If instance has periodic boundary conditions then `-1` is effectively `N-1`. Similarly `N` is the right boundary condition.

To see how the wiring affects the result:

```
>>> ca = RewiredECA(30, size=3)
>>> ca.update([0, 1, 0])
[1, 1, 1]
>>> ca = RewiredECA(30, wiring=[[0,1,3], [1,1,1], [2,1,2]])
>>> ca.update([0, 1, 0])
[1, 0, 1]
```

Parameters

- **code** (*int*) – the 8-bit Wolfram code for the rule
- **boundary** (*tuple, None*) – the boundary conditions for the CA
- **size** (*int or None*) – the number of cells in the lattice
- **wiring** (*list, numpy.ndarray*) – a wiring matrix
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

Raises

- **ValueError** – if both `size` and `wiring` are provided
- **ValueError** – if neither `size` nor `wiring` are provided
- **ValueError** – if `size` is less than 1 (when provided)
- **ValueError** – if `wiring` is not a $3 \times N$ matrix (when provided)
- **ValueError** – if any element of `wiring` is outside the range $[-1,]$‘ (when provided)

code

The Wolfram code of the elementary cellular automaton

Examples

```
>>> reca = RewiredECA(30, size=55)
>>> reca.code
30
>>> reca.code = 45
>>> reca.code
45
>>> reca.code = 256
Traceback (most recent call last):
    ...
ValueError: invalid ECA code
```

Type int

Raises ValueError – if code is not in $\{0, 1, \ldots, 255\}$

boundary

The boundary conditions of the elemenary cellular automaton

Examples

```
>>> reca = RewiredECA(30, size=5)
>>> reca.boundary
>>> reca.boundary = (0,1)
>>> reca.boundary
(0, 1)
>>> reca.boundary = None
>>> reca.boundary
>>> reca.boundary = [0,1]
Traceback (most recent call last):
    ...
TypeError: ECA boundary are neither None nor a tuple
```

Type tuple, None

Raises `ValueError` – if boundary is neither None nor a pair of binary states

`wiring`
The wiring matrix for the rule.

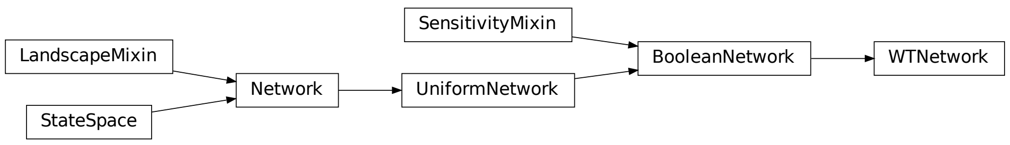
Examples

```
>>> reca = RewiredECA(30, size=4)
>>> reca.wiring
array([[-1,  0,  1,  2],
       [ 0,  1,  2,  3],
       [ 1,  2,  3,  4]])
>>> eca = RewiredECA(30, wiring=[[0,1],[1,1],[-1,-1]])
>>> eca.wiring
array([[ 0,  1],
       [ 1,  1],
       [-1, -1]])
```

Type numpy.ndarray

Weight-Threshold Networks

class neet.boolean.**WTNetwork**(*weights*, *thresholds=None*, *theta=None*, *names=None*, *metadata=None*)
WTNetwork represents weight-threshold boolean network. This type of Boolean network model is common in biology as it represents activating/inhibiting interactions between subcomponents.



In addition to methods inherited from `neet.boolean.BooleanNetwork`, WTNetwork exposes the following attributes

`weights`	The network's square weight matrix.
`thresholds`	The network's threshold vector.
`theta`	The network's activation function.

and static methods:

`read`	Read a network from a pair of node/edge files.
`positive_threshold`	Activate if the stimulus is 0 or greater.
`negative_threshold`	Activate if the stimulus exceeds 0.
`split_threshold`	Activates if the stimulus exceeds 0, maintaining state if it is exactly 0.

At a minimum, WTNetworks accept either a weight matrix or a size. The weight matrix must be square, with the (i, j) element representing the weight on the edge from j-th node to the i-th. If a size is provided, all weights are assumed to be 0.0.

```
>>> WTNetwork(3)
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
>>> WTNetwork([[0, 1, 0], [-1, 0, -1], [-1, 1, 1]])
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
```

Each node has associated with it a threshold value. These thresholds can be provided at initialization. If none are provided, all thresholds are assumed to be 0.0.

```
>>> net = WTNetwork(3, [0.5, 0.0, -0.5])
>>> net.thresholds
array([ 0.5,  0. , -0.5])
>>> WTNetwork([[0, 1, 0], [-1, 0, -1], [-1, 1, 1]], thresholds=[0.5, 0.0, -0.5])
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
```

Finally, every node of the network is assumed to use the same activation function, `theta`. This function, if not provided, is assumed to be `split_threshold()`.

```
>>> net = WTNetwork(3)
>>> net.theta
<function WTNetwork.split_threshold at 0x...>
>>> net = WTNetwork(3, theta=WTNetwork.negative_threshold)
>>> net.theta
<function WTNetwork.negative_threshold at 0x...>
```

This activation function must accept two arguments: the activation stimulus and the current state of the node or network. It should handle two types of arguments:

1. stimulus and state are scalar
2. stimulus and state are vectors (`list` or `numpy.ndarray`)

In case 2, the result should *modify* the state in-place and return the vector.

```
def theta(stimulus, state):
    if isinstance(stimulus, (list, numpy.ndarray)):
        for i, x in enumerate(stimulus):
            state[i] = theta(x, state[i])
```

(continues on next page)

(continued from previous page)

```
        return state
    elif stimulus < 0:
        return 0
    else:
        return state
net = WTNetwork(3, theta=theta)
print(net.theta)
```

```
<function theta at 0x...>
```

As with all `neet.Network` classes, the names of the nodes and network-wide metadata can be provided.

Parameters

- **weights** (*int*, *list*, *numpy.ndarray*) – a weights matrix (rows → targets, columns → sources) or a size
- **thresholds** (*list*, *numpy.ndarray*) – activation thresholds for the nodes
- **theta** (*callable*) – the activation function for all nodes
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

Raises

- **ValueError** – if weights is not a integer or a square matrix
- **ValueError** – if thresholds and weights have inconsistent dimensions
- **ValueError** – if theta is not callable

weights

The network's square weight matrix. The rows and columns are target and source nodes, respectively. That is, the (i, j) element is the weight of the edge from the j-th node to the i-th.

Examples

```
>>> net = WTNetwork(3)
>>> net.weights
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])

>>> net = WTNetwork([[1, 0, 1], [-1, 1, 0], [0, 0, 1]])
>>> net.weights
array([[ 1.,  0.,  1.],
       [-1.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Type numpy.ndarray

thresholds

The network's threshold vector. The i-th element is the threshold for the i-th node.

Examples

```
>>> net = WTNetwork(3)
>>> net.thresholds
array([0., 0., 0.])

>>> net = WTNetwork(3, thresholds=[0, 0.5, -0.5])
>>> net.thresholds
array([ 0. ,  0.5, -0.5])
```

Type numpy.ndarray

theta

The network's activation function. Every node in the network uses this function to determine its next state, based on the simulus it recieves.

```
>>> WTNetwork(3).theta
<function WTNetwork.split_threshold at 0x...>
>>> WTNetwork(3, theta=WTNetwork.negative_threshold).theta
<function WTNetwork.negative_threshold at 0x...>
```

This activation function must accept two arguments: the activation stimulus and the current state of the node or network. It should handle two types of arguments:

1. stimulus and state are scalar
2. stimulus and state are vectors (`list` or `numpy.ndarray`)

In case 2, the result should *modify* the state in-place and return the vector.

```
def theta(stimulus, state):
    if isinstance(stimulus, (list, numpy.ndarray)):
        for i, x in enumerate(stimulus):
            state[i] = theta(x, state[i])
        return state
    elif stimulus < 0:
        return 0
    else:
        return state
net = WTNetwork(3, theta=theta)
print(net.theta)
```

```
<function theta at 0x...>
```

As with all *`neet.Network`* classes, the names of the nodes and network-wide metadata can be provided.

Type callable

static read(*nodes_path*, *edges_path*, *theta=None*, *metadata=None*)

Read a network from a pair of node/edge files.

```
>>> nodes_path = '../neet/boolean/data/s_pombe-nodes.txt'
>>> edges_path = '../neet/boolean/data/s_pombe-edges.txt'
>>> net = WTNetwork.read(nodes_path, edges_path)
>>> net.size
9
>>> net.names
['SK', 'Cdc2_Cdc13', 'Ste9', 'Rum1', 'Slp1', 'Cdc2_Cdc13_active', 'Wee1_Mik1',
↪ 'Cdc25', 'PP']
```

(continues on next page)

(continued from previous page)

```
```

Parameters

- **nodes_path** (*str*) – path to the nodes file
- **edges_path** (*str*) – path to the edges file
- **theta** (*callable*) – the activation function
- **metadata** (*dict*) – metadata dictionary for the network

Returns a *WTNetwork*

static positive_threshold(*values*, *states*)

Activate if the stimulus is 0 or greater. That is, it "leans positive" if the simulus is 0:

$$\theta_p(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0. \end{cases}$$

If `values` and `states` are iterable, then apply the above function to each pair `(x,y)` in `zip(values, states)` and stores the result in `states`.

If `values` and `states` are scalar values, then simply apply the above threshold function to the pair `(values, states)` and return the result.

Examples

```
>>> ys = [0,0,0]
>>> WTNetwork.positive_threshold([1, -1, 0], ys)
[1, 0, 1]
>>> ys
[1, 0, 1]
>>> ys = [1,1,1]
>>> WTNetwork.positive_threshold([1, -1, 0], ys)
[1, 0, 1]
>>> ys
[1, 0, 1]
>>> WTNetwork.positive_threshold(0,0)
1
>>> WTNetwork.positive_threshold(0,1)
1
>>> WTNetwork.positive_threshold(1,0)
1
>>> WTNetwork.positive_threshold(-1,0)
0
```

Parameters

- **values** – the threshold-shifted values of each node
- **states** – the pre-updated states of the nodes

Returns the updated states

static negative_threshold(*values*, *states*)

Activate if the stimulus exceeds 0. That is, it "leans negative" if the simulus is 0:

$$\theta_n(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0. \end{cases}$$

If `values` and `states` are iterable, then apply the above function to each pair `(x,y)` in `zip(values, states)` and stores the result in `states`.

If `values` and `states` are scalar values, then simply apply the above threshold function to the pair `(values, states)` and return the result.

Examples

```
>>> ys = [0,0,0]
>>> WTNetwork.negative_threshold([1, -1, 0], ys)
[1, 0, 0]
>>> ys
[1, 0, 0]
>>> ys = [1,1,1]
>>> WTNetwork.negative_threshold([1, -1, 0], ys)
[1, 0, 0]
>>> ys
[1, 0, 0]
>>> WTNetwork.negative_threshold(0,0)
0
>>> WTNetwork.negative_threshold(0,1)
0
>>> WTNetwork.negative_threshold(1,0)
1
>>> WTNetwork.negative_threshold(1,1)
1
```

Parameters

- **values** – the threshold-shifted values of each node
- **states** – the pre-updated states of the nodes

Returns the updated states

static split_threshold(*values*, *states*)

Activates if the stimulus exceeds 0, maintaining state if it is exactly 0. That is, it is a middle ground between *negative_threshold()* and *positive_threshold()*:

$$\theta_s(x,y) = \begin{cases} 0 & x < 0 \\ y & x = 0 \\ 1 & x > 0. \end{cases}$$

If `values` and `states` are iterable, then apply the above function to each pair `(x,y)` in `zip(values, states)` and stores the result in `states`.

If `values` and `states` are scalar values, then simply apply the above threshold function to the pair `(values, states)` and return the result.

Examples

```
>>> ys = [0,0,0]
>>> WTNetwork.split_threshold([1, -1, 0], ys)
[1, 0, 0]
>>> ys
[1, 0, 0]
>>> ys = [1,1,1]
>>> WTNetwork.split_threshold([1, -1, 0], ys)
[1, 0, 1]
>>> ys
[1, 0, 1]
>>> WTNetwork.split_threshold(0,0)
0
>>> WTNetwork.split_threshold(0,1)
1
>>> WTNetwork.split_threshold(1,0)
1
>>> WTNetwork.split_threshold(1,1)
1
```

Parameters

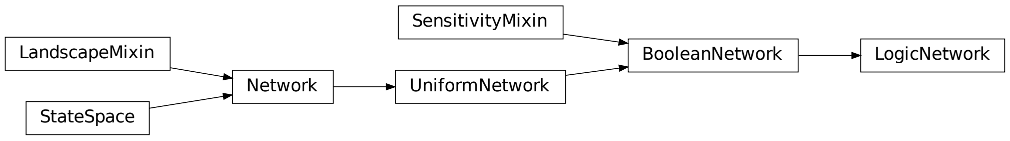
- **`values`** – the threshold-shifted values of each node
- **`states`** – the pre-updated states of the nodes

Returns the updated states

Logical Networks

class `neet.boolean.`**`LogicNetwork`**(*table*, *reduced=False*, *names=None*, *metadata=None*)

LogicNetwork represents a network of logic functions. This type of Boolean network model is common in biological modeling.



In addition to methods inherited from *`neet.boolean.BooleanNetwork`*, LogicNetwork exposes the following attributes

`table`	The network's truth table.

and methods:

is_dependent	Is the target node dependent on the state of source?
reduce_table	Reduce truth table by removing input nodes which have no logic influence from the truth table of each node.
read_table	Read a network from a truth table file.
read_logic	Read a network from a file of logic equations.

At a minimum, LogicNetworks accept a truth table at initialization. A truth table stores a list of tuples, one for each node in order. A tuple of the form `(A, {C1, C2, ...})` at index `i` provides the activation conditions for the node of index `i`. `A` is a tuple marking the indices of the nodes which influence the state of node `i` via logic relations. `{C1, C2, ...}` is a set, each element of which is the collection of binary states of these influencing nodes that would activate node `i`, setting it to `1`. Any other collection of states of nodes in `A` are assumed to deactivate node `i`, setting it to `0`.

`C1`, `C2`, etc. are sequences (`tuple` or `str`) of binary digits, each being the binary state of corresponding node in `A`.

The following network has a single node, which is only activates when it is in the `0` state. That is, it alternates between `0` and `1`.

```
>>> net = LogicNetwork([((0,), {'0'})])
>>> net.size
1
>>> net.table
[((0,), {'0'})]
```

A more complicated network, with three nodes. Here, node `0` activates in the next state whenever node `1` is deactivated; node `1` activates based on the state of nodes `1` and `2`; and node `2` activates based on its own state.

```
>>> net = LogicNetwork([((1,), {'0'}), ((1,2), {'10', '11'}), ((2,), {'1'})])
>>> net.size
3
>>> net.table == [((1,), {'0'}), ((1, 2), {'10', '11'}), ((2,), {'1'})]
True
```

Notice that node `1` will fall into the activated state regardless of what node `2` is doing. In other words, the edge $2 \to 1$ is not a real edge. The table can be reduced to remove such an "fake" edge using the `reduced` argument:

```
>>> net = LogicNetwork([((1,), {'0'}), ((1,2), {'10', '11'}), ((2,), {'1'})])
>>> net.table == [((1,), {'0'}), ((1, 2), {'10', '11'}), ((2,), {'1'})]
True
>>> net = LogicNetwork([((1,), {'0'}), ((1,2), {'10', '11'}), ((2,), {'1'})],␣
↪reduced=True)
>>> net.table == [((1,), {'0'}), ((1,), {'1'}), ((2,), {'1'})]
True
```

Parameters

- **table** (*list, tuple*) – the logic table
- **reduced** (*bool*) – reduce the table
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

Raises

- **TypeError** – if the rows of the table are neither `list` nor `tuple`
- **IndexError** – if a node depends another which doesn't have a row in the table
- **TypeError** – if the truth conditions are neither `list`, `tuple` nor `set`.

table

The network's truth table.

A truth table is a list of tuples, one for each node in order. A tuple of the form `(A, {C1, C2, ...})` at index `i` provides the activation conditions for the node of index `i`. `A` is a tuple marking the indices of the nodes which influence the state of node `i` via logic relations. `{C1, C2, ...}` is a set, each element of which is the collection of binary states of these influencing nodes that would activate node `i`, setting it to `1`. Any other collection of states of nodes in `A` are assumed to deactivate node `i`, setting it to `0`.

`C1`, `C2`, etc. are sequences (`tuple` or `str`) of binary digits, each being the binary state of corresponding node in `A`.

```
>>> from neet.boolean.examples import myeloid
>>> myeloid.table == [((0, 1, 2, 7), {'1000', '1100', '1010'}),
... ((1, 0, 4, 7), {'0010', '1100', '1010', '1110', '0110', '0100', '1000'}),
... ((1,), {'1'}),
... ((1, 4), {'10'}),
... ((1, 3), {'10'}),
... ((1, 7), {'10'}),
... ((6, 1, 2, 5), {'1011', '1100', '1010', '1110', '1101', '1000', '1001'}),
... ((6, 7, 1, 0), {'1000', '1100', '0100'}),
... ((7, 10), {'10'}),
... ((7, 8, 10), {'110'}),
... ((6, 9), {'10'})]
True
```

Type list of tuples of type (list, set)

is_dependent (*target*, *source*)

Is the `target` node dependent on the state of `source`?

```
>>> net = LogicNetwork([((1, 2), {'01', '10'}),
... ((0, 2), {'01', '10', '11'}),
... ((0, 1), {'11'})])
>>> net.is_dependent(0, 0)
False
>>> net.is_dependent(0, 2)
True
```

Parameters

- **target** (*int*) – index of the target node
- **source** (*int*) – index of the source node

Returns whether the target node is dependent on the source

reduce_table ()

Reduce truth table by removing input nodes which have no logic influence from the truth table of each node.

Note: This function introduces the identity function for all nodes which have no inputs. This ensure that every node has a well-defined logical function. The example below demonstrates this with node `1`.

```
>>> net = LogicNetwork([((0,1), {'00', '10'}), ((0,), {'0', '1'})])
>>> net.table == [((0,1), {'00', '10'}), ((0,), {'0', '1'})]
True
>>> net.reduce_table()
>>> net.table == [((1,), {'0'}), ((1,), {'0', '1'})]
True
```

classmethod read_table(*table_path*, *reduced=False*, *metadata=None*)

Read a network from a truth table file.

A logic table file starts with a table title which contains names of all nodes. It is a line marked by `##` at the begining with node names seperated by commas or spaces. This line is required. For artificial network without node names, arbitrary names must be put in place, e.g.:

```
## A B C D
```

Following are the sub-tables of logic conditions for every node. Each sub-table nominates a node and its logically connected nodes in par- enthesis as a comment line:

```
# A (B C)
```

The rest of the sub-table are states of those nodes in parenthesis `(B, C)` that would activate the state of A. States that would deactivate `A` should not be included in the sub-table.

A complete logic table with 3 nodes A, B, C would look like this:

```
## A B C
# A (B C)
1 0
1 1
# B (A)
1
# C (B C A)
1 0 1
0 1 0
0 1 1
```

Custom comments can be added above or below the table title (as long as they are preceeded with more or less than two `#` (e.g. `#` or `###` but not `##`)).

Examples:

```
print(open(MYELOID_TRUTH_TABLE, 'r').read())
```

```
## GATA-2, GATA-1, FOG-1, EKLF, Fli-1, SCL, C/EBPa, PU.1, cJun, EgrNab, Gfi-1
# GATA-2 (GATA-2, GATA-1, FOG-1, PU.1)
1 1 0 0
1 0 1 0
1 0 0 0
# GATA-1 (GATA-1, GATA-2, Fli-1, PU.1)
1 0 0 0
```

(continues on next page)

(continued from previous page)

```
0 1 0 0
0 0 1 0
1 1 0 0
1 0 1 0
0 1 1 0
1 1 1 0
# FOG-1 (GATA-1)
1
...
```

```
>>> net = LogicNetwork.read_table(MYELOID_TRUTH_TABLE)
>>> net.size
11
>>> net.names
['GATA-2', 'GATA-1', 'FOG-1', 'EKLF', 'Fli-1', 'SCL', 'C/EBPa', 'PU.1', 'cJun
↪', 'EgrNab', 'Gfi-1']
>>> net.table ==  [((0, 1, 2, 7), {'1000', '1010', '1100'}),
... ((1, 0, 4, 7), {'0010', '0100', '0110', '1000', '1010', '1100', '1110'}),
... ((1,), {'1'}),
... ((1, 4), {'10'}),
... ((1, 3), {'10'}),
... ((1, 7), {'10'}),
... ((6, 1, 2, 5), {'1000', '1001', '1010', '1011', '1100', '1101', '1110'}),
... ((6, 7, 1, 0), {'0100', '1000', '1100'}),
... ((7, 10), {'10'}),
... ((7, 8, 10), {'110'}),
... ((6, 9), {'10'})]
True
```

Parameters

- **table_path** (*str*) – a path to a table table file
- **reduced** (*bool*) – reduce the table
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

Returns a *LogicNetwork*

classmethod read_logic(*logic_path*, *external_nodes_path=None*, *reduced=False*, *metadata=None*)

Read a network from a file of logic equations.

A logic equations has the form of `A = B AND ( C OR D )`, each term being separated from parantheses and logic operators with at least a space. The optional `external_nodes_path` takes a file that contains nodes in a column whose states do not depend on any nodes. These are considered “external” nodes. Equivalently, such a node would have a logic equation `A = A`, for its state stays on or off unless being set externally.

Examples

```
print(open(MYELOID_LOGIC_EXPRESSIONS, 'r').read())
```

```
GATA-2 = GATA-2 AND NOT ( GATA-1 AND FOG-1 ) AND NOT PU.1
GATA-1 = ( GATA-1 OR GATA-2 OR Fli-1 ) AND NOT PU.1
FOG-1 = GATA-1
EKLF = GATA-1 AND NOT Fli-1
Fli-1 = GATA-1 AND NOT EKLF
SCL = GATA-1 AND NOT PU.1
C/EBPa = C/EBPa AND NOT ( GATA-1 AND FOG-1 AND SCL )
PU.1 = ( C/EBPa OR PU.1 ) AND NOT ( GATA-1 OR GATA-2 )
cJun = PU.1 AND NOT Gfi-1
EgrNab = ( PU.1 AND cJun ) AND NOT Gfi-1
Gfi-1 = C/EBPa AND NOT EgrNab
```

```
>>> net = LogicNetwork.read_logic(MYELOID_LOGIC_EXPRESSIONS)
>>> net.size
11
>>> net.names
['GATA-2', 'GATA-1', 'FOG-1', 'EKLF', 'Fli-1', 'SCL', 'C/EBPa', 'PU.1', 'cJun
↪', 'EgrNab', 'Gfi-1']
>>> net.table ==  [((0, 1, 2, 7), {'1000', '1010', '1100'}),
... ((1, 0, 4, 7), {'0010', '0100', '0110', '1000', '1010', '1100', '1110'}),
... ((1,), {'1'}),
... ((1, 4), {'10'}),
... ((1, 3), {'10'}),
... ((1, 7), {'10'}),
... ((6, 1, 2, 5), {'1000', '1001', '1010', '1011', '1100', '1101', '1110'}),
... ((6, 7, 1, 0), {'0100', '1000', '1100'}),
... ((7, 10), {'10'}),
... ((7, 8, 10), {'110'}),
... ((6, 9), {'10'})]
True
```

Parameters

- **logic_path** (*str*) – path to a file of logial expressions
- **external_nodes_path** (*str*) – a path to a file of external nodes
- **reduced** (*bool*) – reduce the table
- **names** (*seq*) – an iterable object of the names of the nodes in the network
- **metadata** (*dict*) – metadata dictionary for the network

Returns a *LogicNetwork*

Sensitivity Analysis

class neet.boolean.**SensitivityMixin**

SensitivityMixin provides methods for sensitivity analysis. That is, methods to quantify the degree to which perturbations of a network's state propagate and spread. As part of this, we also provide methods for identifying "canalizing edges": edges for which a state of the source node uniquely determines the state of the target regardless of other sources.

sensitivity	Compute the Boolean sensitivity at a given network state.

Continued on next page

Table 11 – continued from previous page

average_sensitivity	Calculate average Boolean network sensitivity, as defined in [Shmulevich2004].
lambdaQ	Compute the sensitivity eigenvalue, λ_Q.
difference_matrix	Compute the difference matrix at a given state.
average_difference_matrix	Compute the difference matrix, averaged over some states.
is_canalizing	Determine whether a given network edge is canalizing.
canalizing_edges	Get the set of all canalizing edges in the network.
canalizing_nodes	Get a set of all nodes with at least one incoming canalizing edge.

The *neet.boolean.BooleanNetwork* class derives from SensitivityMixin to provide sensitivity analysis to all of Neet's Boolean network models.

sensitivity(*state*, *transitions=None*)

Compute the Boolean sensitivity at a given network state.

The sensitivity of a Boolean function f on state vector x is the number of Hamming neighbors of x on which the function value is different than on x, as defined in [Shmulevich2004].

This method calculates the average sensitivity over all N boolean functions, where N is the number of nodes in the network.

Examples

```
>>> s_pombe.sensitivity([0, 0, 0, 0, 0, 1, 1, 0, 0])
1.0
>>> s_pombe.sensitivity([0, 1, 1, 0, 1, 0, 0, 1, 0])
0.4444444444444444
>>> c_elegans.sensitivity([0, 0, 0, 0, 0, 0, 0, 0])
1.75
>>> c_elegans.sensitivity([1, 1, 1, 1, 1, 1, 1, 1])
1.25
```

Optionally, the user can provide a pre-computed array of state transitions to improve performance when this function is repeatedly called.

```
>>> trans = list(map(s_pombe.decode, s_pombe.transitions))
>>> s_pombe.sensitivity([0, 0, 0, 0, 0, 1, 1, 0, 0], transitions=trans)
1.0
>>> s_pombe.sensitivity([0, 1, 1, 0, 1, 0, 0, 1, 0], transitions=trans)
0.4444444444444444
```

Parameters

- **state** (*list, numpy.ndarray*) – a single network state
- **transitions** (*list, numpy.ndarray, None*) – precomputed state transitions (*optional*)

Returns the sensitivity at the provided state

See also:

average_sensitivity()

average_sensitivity(*states=None*, *weights=None*, *calc_trans=True*)

Calculate average Boolean network sensitivity, as defined in [Shmulevich2004].

The sensitivity of a Boolean function f on state vector x is the number of Hamming neighbors of x on which the function value is different than on x.

The average sensitivity is an average taken over initial states.

Examples

```
>>> c_elegans.average_sensitivity()
1.265625
>>> c_elegans.average_sensitivity(states=[[0, 0, 0, 0, 0, 0, 0, 0],
... [1, 1, 1, 1, 1, 1, 1, 1]])
...
1.5
>>> c_elegans.average_sensitivity(states=[[0, 0, 0, 0, 0, 0, 0, 0],
... [1, 1, 1, 1, 1, 1, 1, 1]], weights=[0.9, 0.1])
...
1.7
>>> c_elegans.average_sensitivity(states=[[0, 0, 0, 0, 0, 0, 0, 0],
... [1, 1, 1, 1, 1, 1, 1, 1]], weights=[9, 1])
...
1.7
```

Parameters

- **states** (*list, numpy.ndarray, None*) – The states to average over; all states if `None`
- **weights** (*list, numpy.ndarray, None*) – weights for a weighted average over `states`; all 1.
- **calc_trans** – pre-compute all state transitions; ignored if `states` or `weights` is `None`.

Returns the average sensitivity of `net`

See also:

sensitivity()

lambdaQ(***kwargs*)

Compute the sensitivity eigenvalue, λ_Q. That is, the largest eigenvalue of the sensitivity matrix *average_difference_matrix()*.

This is analogous to the eigenvalue calculated in [Pomerance2009].

Examples

```
>>> s_pombe.lambdaQ()
0.8265021276831896
>>> c_elegans.lambdaQ()
1.263099227661824
```

Returns the sensitivity eigenvalue (λ_Q) of `net`

See also:

average_difference_matrix()

difference_matrix(*state*, *transitions=None*)

Compute the difference matrix at a given state.

For a network with N nodes, with Boolean functions f_i, the difference matrix is a $N \times N$ matrix

$$A_{ij} = f_i(x) \oplus f_i(x \oplus e_j)$$

where e_j is the network state with the j-th node in the 1 state while all others are 0. In other words, the element A_{ij} signifies whether or not flipping the j-th node's state changes the subsequent state of the i-th node.

Examples

```
>>> s_pombe.difference_matrix([0, 0, 0, 0, 0, 0, 0, 0, 0])
array([[0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 1., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 1., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 1.],
       [0., 1., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0.]])
>>> c_elegans.difference_matrix([0, 0, 0, 0, 0, 0, 0, 0])
array([[1., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 1., 0., 1.],
       [0., 0., 0., 0., 0., 1., 1., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1.]])
```

Parameters

- **state** (*list, numpy.ndarray*) – the starting state
- **transitions** (*list, numpy.ndarray, None*) – precomputed state transitions (*optional*)

Returns the difference matrix

See also:

average_difference_matrix()

average_difference_matrix(*states=None*, *weights=None*, *calc_trans=True*)

Compute the difference matrix, averaged over some states.

Examples

```
>>> s_pombe.average_difference_matrix()
array([[0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
         0.    ],
        [0.    , 0.    , 0.25  , 0.25  , 0.25  , 0.    , 0.    , 0.    ,
         0.    ],
        [0.25  , 0.25  , 0.25  , 0.    , 0.    , 0.25  , 0.    , 0.    ,
         0.25  ],
        [0.25  , 0.25  , 0.    , 0.25  , 0.    , 0.25  , 0.    , 0.    ,
         0.25  ],
        [0.    , 0.    , 0.    , 0.    , 0.    , 1.    , 0.    , 0.    ,
         0.    ],
        [0.    , 0.    , 0.0625, 0.0625, 0.0625, 0.    , 0.0625, 0.0625,
         0.    ],
        [0.    , 0.5   , 0.    , 0.    , 0.    , 0.    , 0.5   , 0.    ,
         0.5   ],
        [0.    , 0.5   , 0.    , 0.    , 0.    , 0.    , 0.    , 0.5   ,
         0.5   ],
        [0.    , 0.    , 0.    , 0.    , 1.    , 0.    , 0.    , 0.    ,
         0.    ]])
>>> c_elegans.average_difference_matrix()
array([[0.25  , 0.25  , 0.    , 0.    , 0.    , 0.25  , 0.25  , 0.25  ],
        [0.    , 0.    , 0.5   , 0.5   , 0.    , 0.    , 0.    , 0.    ],
        [0.5   , 0.    , 0.5   , 0.    , 0.5   , 0.    , 0.    , 0.    ],
        [0.    , 0.    , 1.    , 0.    , 0.    , 0.    , 0.    , 0.    ],
        [0.    , 0.3125, 0.3125, 0.3125, 0.3125, 0.3125, 0.    , 0.3125],
        [0.5   , 0.    , 0.    , 0.    , 0.    , 0.5   , 0.5   , 0.    ],
        [1.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ],
        [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.5   , 0.5   ]])
```

Parameters

- **states** (*list, numpy.ndarray, None*) – the states to average over; all states if `None`
- **weights** (*list, numpy.ndarray, None*) – weights for a weighted average over `states`; uniform weighting if `None`
- **calc_trans** (*bool*) – pre-compute all state transitions; ignored if `states` or `weights` is `None`

Returns the difference matrix as a `numpy.ndarray()`.

See also:

difference_matrix()

is_canalizing(*x*, *y*)

Determine whether a given network edge is canalizing.

An edge (y, x) is canalyzing if x's value at $t + 1$ is fully determined when y's value has a particular value at t, regardless of the values of other nodes.

According to (Stauffer 1987):

```
"A rule [...] is called forcing, or canalizing, if at least one of
its :math:`K` arguments has the property that the result of the
function is already fixed if this argument has one particular
value, regardless of the values for the :math:`K-1` other
arguments."  Note that this is a definition for whether a node's
```

(continues on next page)

(continued from previous page)

```
rule is canalizing, whereas this function calculates whether a
specific edge is canalizing.  Under this definition, if a node has
any incoming canalizing edges, then its rule is canalizing.
```

Examples

```
>>> s_pombe.is_canalizing(1, 2)
True
>>> s_pombe.is_canalizing(2, 1)
False
>>> c_elegans.is_canalizing(7, 7)
True
>>> c_elegans.is_canalizing(1, 3)
True
>>> c_elegans.is_canalizing(4, 3)
False
```

Parameters

- **x** (*int*) – target node's index
- **y** (*int*) – source node's index

Returns whether or not the edge `(y,x)` is canalizing; `None` if the edge does not exist

See also:

canalizing_edges(), *canalizing_nodes()*

canalizing_edges()

Get the set of all canalizing edges in the network.

Examples

```
>>> s_pombe.canalizing_edges()
{(1, 2), (5, 4), (0, 0), (1, 3), (4, 5), (5, 6), (5, 7), (1, 4), (8, 4), (5,␣
↪2), (5, 3)}
>>> c_elegans.canalizing_edges()
{(1, 2), (3, 2), (1, 3), (7, 6), (6, 0), (7, 7)}
```

Returns the set of canalizing edges as in the form `(target, source)`

See also:

is_canalizing(), *canalizing_nodes()*

canalizing_nodes()

Get a set of all nodes with at least one incoming canalizing edge.

Examples

```
>>> s_pombe.canalizing_nodes()
{0, 1, 4, 5, 8}
>>> c_elegans.canalizing_nodes()
{1, 3, 6, 7}
```

Returns the set indices of nodes with at least one canalizing input edge

See also:

is_canalizing(), *canalizing_edges()*

Network Conversions

`neet.boolean.conv.`**`wt_to_logic`**`(`*net*`)`

Convert a *neet.boolean.WTNetwork* to a *neet.boolean.LogicNetwork*.

Examples

```
>>> net = wt_to_logic(s_pombe)
>>> isinstance(net, LogicNetwork)
True
>>> numpy.array_equal(net.transitions, s_pombe.transitions)
True
```

Parameters **net** (*neet.boolean.WTNetwork*) – a network to convert

Returns an equivalent *neet.boolean.LogicNetwork*

Example Networks

`neet.boolean.examples.`**`c_elegans = <neet.boolean.wtnetwork.WTNetwork object>`**

A gene regulatory network model of the *S. elegans* cell cycle, as described in [Huang2013].

`neet.boolean.examples.`**`hgf_signaling_in_keratinocytes = <neet.boolean.logicnetwork.LogicNetw`**

A gene regulatory network model of hepatocyte growth-factor induced migration of primary human keratinocytes, as described in [Singh2012].

`neet.boolean.examples.`**`il_6_signaling = <neet.boolean.logicnetwork.LogicNetwork object>`**

A gene regulatory model of interleukin 6 signaling, as described in [Ryll2011].

`neet.boolean.examples.`**`mouse_cortical_7B = <neet.boolean.logicnetwork.LogicNetwork object>`**

A gene regulatory network model for cortical area development in mice, as described in fig. 7B of [Giacomantonio2010].

`neet.boolean.examples.`**`mouse_cortical_7C = <neet.boolean.logicnetwork.LogicNetwork object>`**

A gene regulatory network model for cortical area development in mice, as described in fig. 7C of [Giacomantonio2010].

`neet.boolean.examples.`**`myeloid = <neet.boolean.logicnetwork.LogicNetwork object>`**

A gene regulatory network for the differentiation of myeloid progenitors, as described in [Krumsiek2011].

`neet.boolean.examples.`**`p53_dmg = <neet.boolean.wtnetwork.WTNetwork object>`**

A simplified gene regulatory network model of p53 signaling *with* damage, as described in [Choi2012].

neet.boolean.examples.**p53_no_dmg = <neet.boolean.wtnetwork.WTNetwork object>**
A simplified gene regulatory network model of p53 signaling *without* damage, as described in [Choi2012].

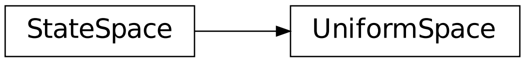
neet.boolean.examples.**s_cerevisiae = <neet.boolean.wtnetwork.WTNetwork object>**
A gene regulatory network model of the *S. cerevisiae* (budding yeast) cell cycle, as described in [Li2004].

neet.boolean.examples.**s_pombe = <neet.boolean.wtnetwork.WTNetwork object>**
A gene regulatory network model of the *S. pombe* (fission yeast) cell cycle, as described in [Davidich2008].

4.8.3 State Spaces

The *neet* module provides the following classes from which all Neet network classes inherit:

StateSpace	StateSpace represents a (potentially in-homogeneous) discrete state space.
UniformSpace	A *StateSpace* with the same number of states in each dimension.



This endows networks with methods for iterating over the states of the network, determining if a state exists in the network, and the ability to encode and decode states as integer values. In other words, these classes provide an interface for accessing the *unstructured* set of states of the network, with no dynamical information.

StateSpace

class neet.**StateSpace**(*shape*)
StateSpace represents a (potentially in-homogeneous) discrete state space. It implements iteration, inclusion testing and methods for encoding and decoding states as integers sutable for array indexing:

size	Get the size of the state space.
shape	Get the shape of the state space.
volume	Get the volume of the state space.
__iter__	Iterate over the states of the state space.
__contains__	Determine if a state is in the state space.
_unsafe_encode	Unsafely encode a state as an integer value.
encode	Encode a state as an integer.
decode	Decode an integer-encoded state into a coordinate list.

StateSpace instances are created from a `shape` array of integer representing the number of discrete states for each dimension of the state space.

Examples

```
>>> StateSpace([2])       # 1-D state space
<neet.statespace.StateSpace object at 0x...>
>>> StateSpace([2,2])     # 2-D uniform state space
<neet.statespace.StateSpace object at 0x...>
>>> StateSpace([2,3,5])   # 3-D inhomogeneous space
<neet.statespace.StateSpace object at 0x...>
```

From the network perspective, each dimension of the state space corresponds to a node of the network. The number of discrete states of that node is the base of the corresponding dimension.

The algorithms implemented by this class are intended to be as generic as possible. This comes at the cost of performance in some cases. This can be dealt with by deriving and overloading the appropriate methods, in particular `_unsafe_encode()`. In fact, the following methods are recommended for overloading:

- `__iter__()`
- `__contains__()`
- `_unsafe_encode()`
- `decode()`

The `encode()` method uses `__contains__()` and `_unsafe_encode()` internally and rarely needs to be overloaded.

Parameters **`shape`** (`list`) – the base of each dimension of the state space

See `UniformSpace`

`size`

Get the size of the state space. That is the number of dimensions.

Examples

```
>>> StateSpace([2]).size
1
>>> StateSpace([2,3,4]).size
3
```

Returns the number of dimensions of the state space

`shape`

Get the shape of the state space. That is the base of each dimension.

Examples

```
>>> StateSpace([2]).shape
[2]
>>> StateSpace([2,3,4]).shape
[2, 3, 4]
```

Returns the shape of the state space

`volume`

Get the volume of the state space. That is the number of states in the space.

Examples

```
>>> StateSpace([2]).volume
2
>>> StateSpace([2,3,4]).volume
24
```

Returns the number of states in the space

__iter__()
Iterate over the states of the state space.

Examples

```
>>> list(StateSpace([2]))
[[0], [1]]
>>> list(StateSpace([2,2]))
[[0, 0], [1, 0], [0, 1], [1, 1]]
>>> list(StateSpace([3,2]))
[[0, 0], [1, 0], [2, 0], [0, 1], [1, 1], [2, 1]]
```

__contains__(*states*)
Determine if a state is in the state space.

Examples

```
>>> space = StateSpace([2])
>>> [0] in space
True
>>> 0 in space
False
```

```
>>> space = StateSpace([3,2])
>>> [2,0] in space
True
>>> [0,2] in space
False
>>> [2,0,0] in space
False
```

_unsafe_encode(*state*)
Unsafely encode a state as an integer value.

Examples

```
>>> space = StateSpace([2,3])
>>> space._unsafe_encode([1,1])
3
```

The resulting numeric encodings must be consistent with the ordering of the states produced by `__iter__()`. This allows necessary for memory-efficient implementations of many algorithms.

```
>>> space = StateSpace([2,3])
>>> list(space)
[[0, 0], [1, 0], [0, 1], [1, 1], [0, 2], [1, 2]]
>>> list(map(space._unsafe_encode, space))
[0, 1, 2, 3, 4, 5]
```

Note: This method is **not** safe. It does not ensure that `state` is in fact in the space; if that's not the case then there are not guaruntees on the output. As such it should only be used in situations where the state is already known to be in the space, e.g. it is a state that was generated by `__iter__()`. This is designed to allow algorithms to utilize state encoding without incurring the cost of consistency checking.

Parameters **`state`** (*`int`*) – the state as a list of coordinates

Returns the state encoded as an integer

See `encode()`, `decode()`

`encode`(*state*)
Encode a state as an integer.

Examples

```
>>> space = StateSpace([2,3])
>>> space.encode([1,1])
3
```

The resulting numeric encodings are consistent with the ordering of the states produced by `__iter__()`.

```
>>> space = StateSpace([2,3])
>>> list(space)
[[0, 0], [1, 0], [0, 1], [1, 1], [0, 2], [1, 2]]
>>> list(map(space.encode, space))
[0, 1, 2, 3, 4, 5]
```

This method is the inverse of the `decode()` method:

```
>>> space = StateSpace([3,2])
>>> space.decode(space.encode([1,1]))
[1, 1]
>>> space.encode(space.decode(3))
3
```

Parameters **`state`** (*`int`*) – the state as a list of coordinates

Returns the state encoded as an integer

See `encode()`, `decode()`

`decode`(*encoded*)
Decode an integer-encoded state into a coordinate list.

Examples

```
>>> space = StateSpace([2,3])
>>> space.decode(3)
[1, 1]
```

The resulting decoded states are consistent with the ordering of the states produced by `__iter__()`.

```
>>> space = StateSpace([2,3])
>>> list(space)
[[0, 0], [1, 0], [0, 1], [1, 1], [0, 2], [1, 2]]
>>> list(map(space.decode, range(0,6)))
[[0, 0], [1, 0], [0, 1], [1, 1], [0, 2], [1, 2]]
```

This method is the inverse of the `encode()` method:

```
>>> space = StateSpace([3,2])
>>> space.decode(space.encode([1,1]))
[1, 1]
>>> space.encode(space.decode(3))
3
```

Parameters **`encoded`** (`int`) – an integer-encoded state

Returns the coordinate list of the decoded state

See `encode()`, `decode()`

UniformSpace

class neet.**UniformSpace**(*size*, *base*)

A `StateSpace` with the same number of states in each dimension. This allows for more efficient implementations of several methods.

UniformSpace instances are created from their `size` and `base`; the number of dimensions and the number of states in each dimension, respectively.

In addition to the methods and attributes exposed by `StateSpace`, the UniformSpace also provides:

`base`	Get the base of the dimensions.

Examples

```
>>> UniformSpace(1, 2) # 1-D unform space with base-2 dimensions
<neet.statespace.UniformSpace object at 0x...>
>>> UniformSpace(2, 2) # 2-D uniform space with base-2 dimensions
<neet.statespace.UniformSpace object at 0x...>
>>> UniformSpace(2, 4) # 2-D uniform space with base-4 dimension
<neet.statespace.UniformSpace object at 0x...>
```

Parameters

- **`size`** (`int`) – the number of dimensions in the space
- **`base`** (`int`) – the number of states in each dimension

See *StateSpace*

base

Get the base of the dimensions.

Examples

```
>>> UniformSpace(2, 3).base
3
```

Returns the base of the space's dimensions

4.8.4 Landscape Analysis

The *neet* module provides the *LandscapeMixin* class from which the *neet.Network* class inherits. This endows all networks with the various methods for computing the various landscape-related properties of the networks, such as *LandscapeMixin.attractors*. These properties are often associated with the *state space* of the network; however, we have opted to provide them via a separate mixin because the *neet.StateSpace* class represents an *unstructured* set of states, with no dynamical information

A key feature of the *LandscapeMixin* is that it is lazy and caches results as they are computed. For example, the attractors of the landscape are computed the first the user requests the *LandscapeMixin.attractors* property, but the result is cached in the *LandscapeMixin.landscape_data* attribute. Subsequent calls simply return the cached data. What's more, many of the properties of the landscape can be determined using almost the exact same algorithm, so whenever one is requested, they are all simultaneously computed. See *LandscapeMixin.expound* for a list of such properties.

LandscapeData

class neet.**LandscapeData**

The LandscapeData class stores the various landscape properties computed in the *LandscapeMixin*. This is used rather an individual properties within *LandscapeMixin* to make it simple for users to extract all of the landscape properties before modifying a network and observing the effects of that change on the landscape.

The following properties are stored in LandscapeData:

LandscapeMixin.transitions	Get the state transitions as an array.
LandscapeMixin.attractors	Get the attractors of the landscape as an array.
LandscapeMixin.attractor_lengths	Get the length of the attractors as an array.
LandscapeMixin.basins	Get the basins of the states as an array.
LandscapeMixin.basin_sizes	Get the sizes of the attractor basins as an array.
LandscapeMixin.basin_entropy	Compute the basin entropy of the landscape [Krawitz2007].
LandscapeMixin.heights	Get the heights of each state in the landscape.
LandscapeMixin.recurrence_times	Get the recurrence time of each state in the landscape.
LandscapeMixin.in_degrees	Get the in-degree of each state in the landscape.

Basic Usage

```
>>> s_pombe.attractors
array([array([76]), array([4]), array([8]), array([12]),
       array([144, 110, 384]), array([68]), array([72]), array([132]),
       array([136]), array([140]), array([196]), array([200]),
       array([204])], dtype=object)
>>> default_landscape = s_pombe.landscape_data

>>> s_pombe.landscape(pin=[0,1]).attractors
array([array([0]), array([1]), array([386, 402, 178, 162]),
       array([387, 403, 179, 163]), array([4]), array([8]), array([12]),
       array([76]), array([65]), array([64]), array([68]), array([72]),
       array([132]), array([136]), array([140]), array([192]),
       array([193]), array([196]), array([200]), array([204])],
      dtype=object)

>>> default_landscape.attractors
array([array([76]), array([4]), array([8]), array([12]),
       array([144, 110, 384]), array([68]), array([72]), array([132]),
       array([136]), array([140]), array([196]), array([200]),
       array([204])], dtype=object)

>>> s_pombe.clear_landscape()
```

LandscapeMixin

class neet.**LandscapeMixin**

The LandscapeMixin class represents the structure and topology of the "landscape" of state transitions. That is, it is the state space together with information about state transitions and the topology of the state transition graph.

The LandscapeMixin class exposes the following methods:

landscape	Setup the landscape.
clear_landscape	Clear the landscape's data and graph from memory.
landscape_data	Get the *LandscapeData* object.
transitions	Get the state transitions as an array.
attractors	Get the attractors of the landscape as an array.
attractor_lengths	Get the length of the attractors as an array.
basins	Get the basins of the states as an array.
basin_sizes	Get the sizes of the attractor basins as an array.
basin_entropy	Compute the basin entropy of the landscape [Krawitz2007].
heights	Get the heights of each state in the landscape.
recurrence_times	Get the recurrence time of each state in the landscape.
in_degrees	Get the in-degree of each state in the landscape.
trajectory	Compute the trajectory from a given state.
timeseries	Compute a time series from all states.
landscape_graph	Construct a `networkx.DiGraph` of the state transitions.
draw_landscape_graph	Draw the state transition graph.

Continued on next page

Table 16 – continued from previous page

expound	Compute all cached data.

landscape(*index=None*, *pin=None*, *values=None*)

Setup the landscape.

Prepares the landscape for computation of the various properties, specifying which nodes will be updated (`index`), pinned (`pin`) or set to a particular state (`values`). In particular, it computes the state transitions of the network and prepares private variables for a subsequent call to *expound()*, *landscape_graph()*, etc…

This function is implicitly called with no arguments by the various landscape accessors if it has not already been called. This is intended as a convenience since most of the time the user would do this anyway.

This function implicitly calls *clear_landscape*, so make sure to create a reference to *landscape_data* if landscape information has previously been compute and you wish to keep it around.

Basic Usage

```
>>> s_pombe.landscape_data.transitions
>>> s_pombe.landscape()
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
>>> len(s_pombe.landscape_data.transitions)
512
```

Pinning States

```
# Prevents all states from transitioning
>>> s_pombe.landscape(pin = range(s_pombe.size))
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
>>> np.array_equal(s_pombe.landscape_data.transitions, range(s_pombe.volume))
True
>>> s_pombe.clear_landscape()
```

Overriding Node States

```
# Forces all states to transition to 0
>>> s_pombe.landscape(values={i: 0 for i in range(s_pombe.size)})
<neet.boolean.wtnetwork.WTNetwork object at 0x...>
>>> np.all(s_pombe.landscape_data.transitions == 0)
True
>>> s_pombe.clear_landscape()
```

Parameters

- **index** – the index to update (or None)
- **pin** – the indices to pin during update (or None)
- **values** – a dictionary of index-value pairs to set after update

Returns `self`

clear_landscape()

Clear the landscape's data and graph from memory.

landscape_data

Get the *LandscapeData* object.

The *LandscapeData* object contains any cached attractor landscape information generated by a call to *expound()*.

transitions

Get the state transitions as an array. Each element of the array is the next (encoded) state of the system starting from the initial state equal to the index. For example, if

```
>>> net.transitions
array([ 0, 3, 1, 2 ])
```

then state `0` will transition to `0`, `1` to `3`, etc… Be aware that if *landscape()* has not been called, this method will call it.

Basic Usage

```
>>> s_pombe.transitions
array([  2,   2, 130, 130,   4,   0, 128, 128,   8,   0, 128, 128,  12,
         0, 128, 128, 256, 256, 384, 384, 260, 256, 384, 384, 264, 256,
       ...
       208, 208, 336, 336, 464, 464, 340, 336, 464, 464, 344, 336, 464,
       464, 348, 336, 464, 464])
```

Pinned States

A preceding call to *landscape()* can, for example, pin specific nodes to their current state, thus affecting the state transitions.

```
>>> s_pombe.landscape(pin = [0]).transitions
array([  2,   3, 130, 131,   4,   1, 128, 129,   8,   1, 128, 129,  12,
         1, 128, 129, 256, 257, 384, 385, 260, 257, 384, 385, 264, 257,
       ...
       208, 209, 336, 337, 464, 465, 340, 337, 464, 465, 344, 337, 464,
       465, 348, 337, 464, 465])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray` of state transitions

attractors

Get the attractors of the landscape as an array. Each element of the array is an attractor cycle, each of which is an array of states in the cycle. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.attractors
array([array([76]), array([4]), array([8]), array([12]),
       array([144, 110, 384]), array([68]), array([72]), array([132]),
```

(continues on next page)

(continued from previous page)

```
        array([136]), array([140]), array([196]), array([200]),
        array([204])], dtype=object)
```

Update Only a Single Node

A preceding call to *landscape()* can, for example, specify which nodes will be updated in the process of computing the attractors. For example, we can allow only the first node of the state to be updated.

```
>>> s_pombe.landscape(index=0).attractors
array([[  0],
       [  2],
       [  4],
      ...
       [506],
       [508],
       [510]])
>>> s_pombe.clear_landscape()
```

Returns a numpy.ndarray of attractor cycles, each of which is an array of encoded states

attractor_lengths

Get the length of the attractors as an array. The array is indexed by the basin number. The order of the attractor lengths is the same as in *attractors*. For example,

```
>>> net.attractors
array([ array([0,1]), array([1]) ]
>>> net.attractor_lengths
array([2, 1])
```

If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.attractor_lengths
array([1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1])
```

Pinned States

A preceding call to *landscape()* can pin specific nodes to their current state, thus affecting the attractor lengths.

```
>>> s_pombe.landscape(pin = [0]).attractor_lengths
array([1, 6, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1])
>>> s_pombe.clear_landscape()
```

Returns a numpy.ndarray of the lengths of the attractors

basins

Get the basins of the states as an array. Each index of the array is an encoded state and the corresponding value is the attractor basin in which it resides. The attractor basins are integers which can be used to index the *attractors* array, providing the attractor cycle for the base. For example, if

```
>>> net.basins
array([ 0, 1, 2, 1 ])
>>> net.attractors
array([ array([0]), array([1]), array([2]) ])
```

then the states 1 and 3 are both in the attractor basin which attracts to the fixed-point 1. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.basins
array([ 0,   0,   0,   0,   1,   0,   0,   0,   2,   0,   0,   0,   3,   0,   0,   0,   0,
        0,   4,   4,   0,   0,   4,   4,   0,   0,   4,   4,   0,   0,   4,   4,   4,   4,
        ...
        0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
        0,   0])
```

Resetting Node States

A preceding call to *landscape()* can, for example, specify that specific nodes are reset to a particular value after the updating the. For example, we can force the first and second nodes to 0, thus affecting the basins.

```
>>> s_pombe.landscape(values={0: 0, 1: 0}).basins
array([ 0,   0,   1,   1,   2,   0,   1,   1,   3,   0,   1,   1,   4,   0,   1,   1,   1,
        1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
      ...
        1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
        1,   1])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray` of each state's attractor basin

basin_sizes
Get the sizes of the attractor basins as an array. The array is indexed by the basin number. The order of the basin sizes is the same as in *attractors*. For example, if

```
>>> net.attractors
array([ array([0,1]), array([3,6]) ]
>>> net.basin_sizes
array([ 5, 3 ])
```

then the attractor `[0, 1]` has a basin size of 5 with the remaining states in the other attractor's basin. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.basin_sizes
array([378,   2,   2,   2, 104,   6,   6,   2,   2,   2,   2,   2,   2])
```

Pinning States

A preceding call to *landscape()* can specify that some of the nodes are not updated, say the first two.

```
>>> s_pombe.landscape(pin=[0,1]).basin_sizes
array([  1,   4, 128, 128,   1,   1,   1, 114, 120,   1,   1,   1,   1,
         1,   1,   1,   4,   1,   1,   1])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray` of each attractor's basin size

basin_entropy

Compute the basin entropy of the landscape [Krawitz2007]. That is the Shannon entropy (in bits) of the distribution of basin sizes. For example,

```
>>> net.basin_sizes
array([6, 2])
>>> net.basin_entropy
0.8112781244591328
```

which is $-\frac{6}{8}\log_2\frac{6}{8}) - \frac{2}{8}\log_2\frac{2}{8})$. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.basin_entropy
1.2218888...
```

Pinning States

A preceding call to *landscape()* can specify that some of the nodes are not updated, say the first two.

```
>>> s_pombe.landscape(pin=[0,1]).basin_entropy
2.328561849437885
>>> s_pombe.clear_landscape()
```

Returns basin entropy in bits

heights

Get the heights of each state in the landscape. That is the fewest number of time steps from that state to a state in it's attractor cycle, as an array. Each index of the array is an encoded state, and the corresponding value is the height. For example, if

```
>>> net.heights
array([ 3, 0, 1, ... ])
```

then it will take 3 time steps for the state `0` to reach an attractor state while state `1` **is** an attractor state'. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.heights
array([7, 7, 6, 6, 0, 8, 6, 6, 0, 8, 6, 6, 0, 8, 6, 6, 8, 8, 1, 1, 2, 8,
       1, 1, 2, 8, 1, 1, 2, 8, 1, 1, 2, 2, 2, 2, 9, 9, 1, 1, 9, 9, 1, 1,
       ...
       3, 9, 9, 9, 3, 9, 9, 9, 3, 9, 9, 9, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3])
```

Resetting Node States

A preceding call to *landscape()* can specify that specific nodes are reset to a particular value after the updating the. For example, we can force the first and second nodes to `0`, thus affecting the basins.

```
>>> s_pombe.landscape(values={0: 0, 1: 0}).heights
array([0, 1, 6, 6, 0, 1, 6, 6, 0, 1, 6, 6, 0, 1, 6, 6, 2, 2, 5, 5, 2, 2,
       5, 5, 2, 2, 5, 5, 2, 2, 5, 5, 3, 3, 6, 6, 3, 3, 6, 6, 3, 3, 6, 6,
      ...
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray`, each value of which is the height of the indexing state

recurrence_times

Get the recurrence time of each state in the landscape. That is the number of time steps from that state after which *some* state is repeated, as an array. Each index of the array is an encoded state, and the corresponding value is the recurrence time of that state. For example, if

```
>>> net.recurrent_times
array([ 3, 10, 0, ... ])
```

then a state will be seen at least twice if the `0` state is updated more than 3 times. The `2` state is a fixed-point attractor state as updating even once will repeat a state. If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.recurrence_times
array([7, 7, 6, 6, 0, 8, 6, 6, 0, 8, 6, 6, 0, 8, 6, 6, 8, 8, 3, 3, 2, 8,
       3, 3, 2, 8, 3, 3, 2, 8, 3, 3, 4, 4, 4, 4, 9, 9, 3, 3, 9, 9, 3, 3,
       ...
       3, 9, 9, 9, 3, 9, 9, 9, 3, 9, 9, 9, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3])
```

Resetting Node States

A preceding call to *landscape()* can specify that specific nodes are reset to a particular value after the updating the. For example, we can force the first and second nodes to `0`, thus affecting the basins.

```
>>> s_pombe.landscape(pin=[0,1]).recurrence_times
array([0, 0, 5, 5, 0, 1, 5, 5, 0, 1, 5, 5, 0, 1, 5, 5, 2, 2, 4, 4, 2, 2,
       4, 4, 2, 2, 4, 4, 2, 2, 4, 4, 3, 3, 5, 5, 3, 3, 5, 5, 3, 3, 5, 5,
       ...
       3, 3, 5, 5, 3, 3, 5, 5, 3, 3, 5, 5, 3, 3, 8, 8, 3, 3, 8, 8, 3, 3,
       8, 8, 3, 3, 8, 8])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray` of recurrence times, one for each state

`in_degrees`

Get the in-degree of each state in the landscape. That is the number of states which transition to that state in a single time step, as a array. Each index of the array is an encoded state, and the corresponding value is the number of preceding states. For example, if

```
>>> net.in_degrees
array([ 5, 2, 0, 0, ... ]
```

then 5 states transition to the `0` state in a single time step, while states `2` and `3` are in the Garden of Eden. If *`landscape()`* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.in_degrees
array([ 6,  0,  4,  0,  2,  0,  0,  0,  2,  0,  0,  0,  2,  0,  0,  0, 12,
        0,  4,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        ...
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0])
```

Pinning States

A preceding call to *`landscape()`* can specify that some of the nodes are not updated, say nodes `7` and `8`.

```
>>> s_pombe.landscape(pin=[7,8]).in_degrees
array([36,  0,  6,  0,  2,  0,  0,  0,  2,  0,  0,  0,  2,  0,  0,  0, 42,
        0,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
      ...
        0,  1,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0])
>>> s_pombe.clear_landscape()
```

Returns a `numpy.ndarray` of the in-degree of each state

`trajectory`(*init*, *timesteps=None*, *encode=None*)

Compute the trajectory from a given state.

This method computes a trajectory from `init` to the last before the trajectory begins to repeat. If `timesteps` is provided, then the trajectory will have a length of `timesteps + 1` regardless of repeated states. The `encode` argument forces the states in the trajectory to be either encoded or not. When `encode is None`, whether or not the states of the trajectory are encoded is determined by whether or not the initial state (`init`) is provided in encoded form.

Note that when `timesteps is None`, the length of the resulting trajectory should be one greater than the recurrence time of the state.

If *landscape()* has not been called, this method will implicitly call it. Otherwise, it respects any settings provided by such a call.

Basic Usage

```
>>> s_pombe.trajectory([1,0,0,1,0,1,1,0,1])
[[1, 0, 0, 1, 0, 1, 1, 0, 1], ... [0, 0, 1, 1, 0, 0, 1, 0, 0]]

>>> s_pombe.trajectory([1,0,0,1,0,1,1,0,1], encode=True)
[361, 80, 320, 78, 128, 162, 178, 400, 332, 76]

>>> s_pombe.trajectory(361)
[361, 80, 320, 78, 128, 162, 178, 400, 332, 76]

>>> s_pombe.trajectory(361, encode=False)
[[1, 0, 0, 1, 0, 1, 1, 0, 1], ... [0, 0, 1, 1, 0, 0, 1, 0, 0]]

>>> s_pombe.trajectory(361, timesteps=5)
[361, 80, 320, 78, 128, 162]

>>> s_pombe.trajectory(361, timesteps=10)
[361, 80, 320, 78, 128, 162, 178, 400, 332, 76, 76]
```

Parameters

- **init** (*int or seq*) – the initial state
- **timesteps** (*int or None*) – the number of time steps to include in the trajectory
- **encode** (*bool or None*) – whether to encode the states in the trajectory

Returns a list whose elements are subsequent states of the trajectory

Raises

- **ValueError** – if `init` an empty array
- **ValueError** – if `timesteps` is less than 1

timeseries(*timesteps*)

Compute a time series from all states.

This method computes a 3-dimensional array elements are the states of each node in the network. The dimensions of the array are indexed by, in order, the node, the initial state and the time step.

If *landscape()* has not been called, this method will implicitly call it. Otherwise, it respects any settings provided by such a call.

Basic Usage

```
>>> s_pombe.timeseries(5)
array([[[0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
        ...,
        [1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0]],

       [[0, 1, 1, 1, 1, 0],
        [0, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 0, 0],
        ...,
        [0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0]],

       ...

       [[0, 0, 1, 1, 1, 1],
        [0, 0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1, 0],
        ...,
        [1, 0, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 1, 1],
        ...,
        [1, 1, 1, 0, 0, 0],
        [1, 1, 1, 0, 0, 0],
        [1, 1, 1, 0, 0, 0]]])
```

Parameters **`timesteps`** (`int`) – the number of timesteps to evolve the system

Returns a 3-D array of node states

Raises **`ValueError`** – if `timesteps` is less than 1

`landscape_graph`(***kwargs*)

Construct a `networkx.DiGraph` of the state transitions.

If *`landscape()`* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.landscape_graph()
<networkx.classes.digraph.DiGraph object at 0x...>
```

Parameters **`kwargs`** – kwargs to pass to `networkx.DiGraph`

Returns a `networkx.DiGraph` representing the state transition graph of the landscape

`draw_landscape_graph`(*graphkwargs={}*, *pygraphkwargs={}*)

Draw the state transition graph.

This method requires the optional dependency pygraphviz, which can be installed via `pip`. Be aware that `pygraphviz` requires native binaries of Graphviz which **cannot** be installed via pip.

If *landscape()* has not been called, this method will implicitly call it.

Basic Usage

```
>>> s_pombe.draw_landscape_graph()
```

Parameters

- **graphkwargs** – kwargs to pass to *landscape_graph*
- **pygraphkwargs** – kwargs to pass to *view_pygraphviz*

expound()

Compute all cached data.

This function performs the bulk of the calculations that the LandscapeMixin is concerned with. Most of the properties in this class are computed by this function whenever *any one* of them is requested and the results are cached. The advantage of this is that it saves computation time; why traverse the state space for every property call when you can do it all at once? The downside is that the cached results may use a good bit more memory. This is a trade-off that we are willing to make for now.

The properties that are computed by this function include:

attractors	Get the attractors of the landscape as an array.
attractor_lengths	Get the length of the attractors as an array.
basins	Get the basins of the states as an array.
basin_sizes	Get the sizes of the attractor basins as an array.
basin_entropy	Compute the basin entropy of the landscape [Krawitz2007].
heights	Get the heights of each state in the landscape.
recurrence_times	Get the recurrence time of each state in the landscape.
in_degrees	Get the in-degree of each state in the landscape.

4.8.5 Information Analysis

The *neet* provides the *Information* class to compute various information measures over the dynamics of discrete-state network models.

The core information-theoretic computations are supported by the PyInform package.

class neet.**Information**(*net*, *k*, *timesteps*)

A class to represent the k-history informational architecture of a network.

An Information is initialized with a network, a history length, and time series length. A time series of the desired length is computed from each initial state of the network, and used populate probability distributions over the state transitions of each node. From there any number of information or entropy measures may be applied.

The Information class provides three public attributes:

net	The network over which to compute the various information measures

Continued on next page

Table 18 – continued from previous page

k	The history length to use to compute the various information measures
timesteps	The time series length to use to compute the various information measures

During following measures can be computed and cached:

active_information	Get the local or average active information.
entropy_rate	Get the local or average entropy rate.
mutual_information	Get the local or average mutual information.
transfer_entropy	Get the local or average transfer entropy.

Examples

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.active_information()
array([0.        , 0.4083436 , 0.62956679, 0.62956679, 0.37915718,
       0.40046165, 0.67019615, 0.67019615, 0.39189127])
```

Parameters

- **net** (neet.Network) – the network to analyze
- **k** (*int*) – the history length
- **timesteps** (*int*) – the number of timesteps to evaluate the network

net

The network over which to compute the various information measures

Note: The cached internal state of the *Information* instances, namely any pre-computed time series and information measures, is cleared when the network is changed.

Type *neet.Network*

k

The history length to use to compute the various information measures

Note: The cached internal state of the *Information* instances, namely any pre-computed time series and information measures, is cleared when the history length is changed.

Type int

timesteps

The time series length to use to compute the various information measures

Note: The cached internal state of the *Information* instances, namely any pre-computed time series and information measures, is cleared when the number of time steps is changed.

Type int

active_information(*local=False*)

Get the local or average active information.

Active information (AI) was introduced in [Lizier2012] to quantify information storage in distributed computation. AI is defined in terms of a temporally local variant

$$a_{X,i}(k) = \log_2 \frac{p(x_i^{(k)}, x_{i+1})}{p(x_i^{(k)})p(x_{i+1})}$$

where the probabilites are constructed emperically from an *entire* time series. From this local variant, the temporally global active information is defined as

$$A_X(k) = \langle a_{X,i}(k)\rangle_i = \sum_{x_i^{(k)}, x_{i+1}} p(x_i^{(k)}, x_{i+1}) \log_2 \frac{p(x_i^{(k)}, x_{i+1})}{p(x_i^{(k)})p(x_{i+1})}.$$

Examples

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.active_information()
array([0.        , 0.4083436 , 0.62956679, 0.62956679, 0.37915718,
       0.40046165, 0.67019615, 0.67019615, 0.39189127])
>>> lais = arch.active_information(local=True)
>>> lais[1]
array([[0.13079175, 0.13079175, 0.13079175, ..., 0.13079175, 0.13079175,
        0.13079175],
       [0.13079175, 0.13079175, 0.13079175, ..., 0.13079175, 0.13079175,
        0.13079175],
       ...,
       [0.13079175, 0.13079175, 0.13079175, ..., 0.13079175, 0.13079175,
        0.13079175],
       [0.13079175, 0.13079175, 0.13079175, ..., 0.13079175, 0.13079175,
        0.13079175]])
>>> np.mean(lais[1])
0.4083435...
```

Parameters **local** (*bool*) – whether to return local (True) or global active information

Returns a numpy.ndarray containing the (local) active information for every node in the network

entropy_rate(*local=False*)

Get the local or average entropy rate.

Entropy rate quantifies the amount of information need to describe a random variable — the state of a node in this case — given observations of its k-history. In other words, it is the entropy of the time series of a node's state conditioned on its k-history. The time-local entropy rate

$$h_{X,i}(k) = \log_2 \frac{p(x_i^{(k)}, x_{i+1})}{p(x_i^{(k)})}$$

can be averaged to obtain the global entropy rate

$$H_X(k) = \langle h_{X,i}(k)\rangle_i = \sum_{x_i^{(k)}, x_{i+1}} p(x_i^{(k)}, x_{i+1}) \log_2 \frac{p(x_i^{(k)}, x_{i+1})}{p(x_i^{(k)})}.$$

Examples

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.entropy_rate()
array([0.        , 0.01691208, 0.07280268, 0.07280268, 0.05841994,
       0.02479402, 0.03217332, 0.03217332, 0.08966941])
>>> ler = arch.entropy_rate(local=True)
>>> ler[4]
array([[0.        , 0.        , 0.        , ..., 0.00507099, 0.00507099,
        0.00507099],
       [0.        , 0.        , 0.        , ..., 0.00507099, 0.00507099,
        0.00507099],
       ...,
       [0.        , 0.29604946, 0.00507099, ..., 0.00507099, 0.00507099,
        0.00507099],
       [0.        , 0.29604946, 0.00507099, ..., 0.00507099, 0.00507099,
        0.00507099]])
```

Parameters **local** (*bool*) – whether to return local (True) or global entropy rate

Returns a numpy.ndarray containing the (local) entropy rate for every node in the network

transfer_entropy(*local=False*)

Get the local or average transfer entropy.

Transfer entropy (TE) was introduced by [Schreiber2000] to quantify information transfer between an information source and destination, in this case a pair of nodes, condition out their shared history effects. TE is defined in terms of a time-local variant

$$t_{X\to Y,i}(k) = \log_2 \frac{p(y_{i+1}, x_i \mid y_i^{(k)})}{p(y_{i+1} \mid y_i^{(k)})p(x_i \mid y_i^{(k)})}$$

Time averaging defines the global transfer entropy

$$T_{Y\to X}(k) = \langle t_{X\to Y,i}(k)\rangle_i$$

Examples

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.transfer_entropy()
array([[0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.05137046, 0.05137046, 0.05841994,
        0.        , 0.01668983, 0.01668983, 0.0603037 ],
       ...,
       [0.        , 0.        , 0.00603879, 0.00603879, 0.04760206,
        0.02479402, 0.00298277, 0.        , 0.04892709],
       [0.        , 0.        , 0.07280268, 0.07280268, 0.        ,
        0.        , 0.03217332, 0.03217332, 0.        ]])

>>> lte = arch.transfer_entropy(local=True)
>>> lte[4,3]
array([[-1.03562391,  1.77173101,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [-1.03562391,  1.77173101,  0.        , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
         0.        ,  0.        ],
        [ 1.77173101,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])
```

The first and second indices of the resulting arrays are the source and target nodes, respectively.

Parameters **`local`** (*`bool`*) – whether to return local (True) or global transfer entropy

Returns a `numpy.ndarray` containing the (local) transfer entropy for every pair of nodes in the network

`mutual_information`(*local=False*)

Get the local or average mutual information.

Mutual information is a measure of the amount of mutual dependence (correlation) between two random variables — nodes in this case. The time-local mutual information

$$i_i(X,Y) = -\log_2 \frac{p(x_i,y_i)}{p(x_i)p(y_i)}$$

can be time-averaged to define the standard mutual information

$$I(X,Y) = -\sum_{x_i,y_i} p(x_i,y_i) \log_2 \frac{p(x_i,y_i)}{p(x_i)p(y_i)}.$$

Examples

```
>>> arch = Information(s_pombe, k=5, timesteps=20)
>>> arch.mutual_information()
array([[0.16232618, 0.01374672, 0.00428548, 0.00428548, 0.01340937,
        0.01586238, 0.00516987, 0.00516987, 0.01102766],
       [0.01374672, 0.56660996, 0.00745714, 0.00745714, 0.00639113,
        0.32790848, 0.0067609 , 0.0067609 , 0.00468342],
       ...,
       [0.00516987, 0.0067609 , 0.4590254 , 0.4590254 , 0.17560769,
        0.00621124, 0.49349527, 0.80831657, 0.10390475],
       [0.01102766, 0.00468342, 0.12755745, 0.12755745, 0.01233356,
        0.00260667, 0.10390475, 0.10390475, 0.63423835]])
>>> lmi = arch.mutual_information(local=True)
>>> lmi[4,3]
array([[-0.67489772, -0.67489772, -0.67489772, ...,  0.18484073,
         0.18484073,  0.18484073],
       [-0.67489772, -0.67489772, -0.67489772, ...,  0.18484073,
         0.18484073,  0.18484073],
       ...,
       [-2.89794147,  1.7513014 ,  0.18484073, ...,  0.18484073,
         0.18484073,  0.18484073],
       [-2.89794147,  1.7513014 ,  0.18484073, ...,  0.18484073,
         0.18484073,  0.18484073]])
```

Parameters **local** (*bool*) – whether to return local (True) or global mutual information

Returns a `numpy.ndarray` containing the (local) mutual information for every pair of nodes in the network

4.8.6 Drawing Utilities

Utilities for drawing Neet objects and graph representations.

`neet.draw.`**`view_pygraphviz`**(*G*, *edgelabel=None*, *prog='dot'*, *args="*, *suffix="*, *path=None*, *display_image=True*)

Views the graph G using the specified layout algorithm.

This is a modified version of `view_pyagraphviz` from `networkx.drawing.nx_agraph` to allow display toggle.

Original copyright:

```
Copyright (C) 2004-2019 by
    Aric Hagberg <hagberg@lanl.gov>
    Dan Schult <dschult@colgate.edu>
    Pieter Swart <swart@lanl.gov>
All rights reserved. BSD license.
Author: Aric Hagberg (hagberg@lanl.gov)
```

Parameters

- **G** (*networkx.Graph or networkx.DiGraph*) – the graph to draw
- **edgelabel** (*str, callable or None*) – If a string, then it specifes the edge attribute to be displayed on the edge labels. If a callable, then it is called for each edge and it should return the string to be displayed on the edges. The function signature of *edgelabel* should be edgelabel(data), where *data* is the edge attribute dictionary.
- **prog** (*str*) – Name of Graphviz layout program.
- **args** (*str*) – Additional arguments to pass to the Graphviz layout program.
- **suffix** (*str*) – If *filename* is None, we save to a temporary file. The value of *suffix* will appear at the tail end of the temporary filename.
- **path** (*str or None*) – The filename used to save the image. If None, save to a temporary file. File formats are the same as those from pygraphviz.agraph.draw.

Returns the filename of the generated image, and a `PyGraphviz` graph instance

Note: If this function is called in succession too quickly, sometimes the image is not displayed. So you might consider time.sleep(.5) between calls if you experience problems.

4.8.7 Custom Exceptions

Exceptions are the key mechanism for handling undesirable program state. Whenever Neet encounters a problem, it raises an exception of some variety. Whenever possible, we have preferred to use builtin exception classes, e.g. `ValueError`, `IndexError`, etc… For cases that aren't really covered by a builtin exception class, we've created subclasses of the standard library's `Exception` to report those errors.

FormatError

class neet.exceptions.**FormatError**

An error class to report when a configuration or data file is improperly formatted.

4.9 References

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[Choi2012] Choi, M., Shi, J., Jung, S. H., Chen, X., and Cho, K. H. Cho "Attractor landscape analysis reveals feedback loops in the p53 network that control the cellular response to DNA damage," *Sci. Signal.*, vol. 5, no. 251, p. ra83 (2012) doi:10.1126/scisignal.2003363.

[Davidich2008] Davidich, M. I., and Bornholdt, S. "Boolean network model predicts cell cycle sequence of fission yeast," *PLoS One* **3**, vol. 3, no. 2, p. e1672 (2008) doi:10.1371/journal.pone.0001672.

[Giacomantonio2010] Giacomantonio, C. E., and Goodhill, G. J. "A Boolean model of the gene regulatory network underlying Mammalian cortical area development," *PLoS Comput. Biol.* vol. 6, no. 9 (2010) doi:10.1371/journal.pcbi.1000936.

[Huang2013] Huang, X., Chen, L., Chim, H., Chan, L. L. H., Zhao, Z., and Yan, H. "Boolean genetic network model for the control of C. elegans early embryonic cell cycles," *Biomed. Eng. Online*, vol. 12 Suppl 1, p. S1, (2013) doi:10.1186/1475-925X-12-S1-S1.

[Krawitz2007] Krawitz, P. and Shmulevich, I. "Basin Entropy in Boolean Network Ensembles" *Phys. Rev. Lett.*, vol. 98, 158701 (2007) doi:10.1103/PhysRevLett.98.158701.

[Krumsiek2011] Krumsiek, J., Marr, C., Schroeder, T., and Theis, F. J. "Hierarchical differentiation of myeloid progenitors is encoded in the transcription factor network," *PLoS One*, vol. 6, no. 8, p. e22649, (2011) doi:10.1371/journal.pone.0022649.

[Li2004] Li, F., Long, T., Lu, Y., Ouyang, Q., and Tang, C. "The yeast cell-cycle network is robustly designed," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 101, no. 14, pp. 4781-4786 (2004) doi:10.1073/pnas.0305937101.

[Lizier2012] Lizier, J. T., Prokopenko, M., and Zomaya, A. Y., "Local measures of information storage in complex distributed computation" Information Sciences, vol. 208, pp. 39-54 (2012) doi:10.1016/j.ins.2012.04.016.

[Schreiber2000] T. Schreiber, "Measuring information transfer", *Phys. Rev. Lett* vol. 85, no. 2, pp.461-464 (2000). doi:10.1103/PhysRevLett.85.461.

[Pomerance2009] Pomerance, A., Ott, E., Girvan E., and Losert, W. "The Effect of Network Topology on the Stability of Discrete State Models of Genetic Control." *Proc. Natl. Acad. Sci. USA* **106** (2009): 8209-14. doi:10.1073/pnas.0900142106.

[Ryll2011] Ryll, A., Samaga, R., Schaper, F., Alexopoulos, L. G., Klamt, S. "Large-scale network models of IL-1 and IL-6 signalling and their hepatocellular specification," *Mol. Biosyst.*, vol. 7, no. 12, pp. 3253-3270, (2011) doi:10.1039/c1mb05261f.

[Shmulevich2004] Shmulevich, I., and Kauffman, S. A. “Activities and sensitivities in Boolean network models.” *Phys. Rev. Lett.* **93**, 48701 (2004) doi:10.1103/PhysRevLett.93.048701.

[Singh2012] Singh, A., Nascimento, J. M., Kowar, S., Busch, H., and Boerries, M. “Boolean approach to signalling pathway modelling in HGF-induced keratinocyte migration,” *Bioinformatics*, vol. 28, no. 18, pp. i495-i501, (2012) doi:10.1093/bioinformatics/bts410.

Python Module Index

Index