bindsnet Documentation

Daniel Saunders, Hananel Hazan

Dec 02, 2022

Contents:

BindsNET is built on top of the PyTorch deep learning platform. It is used for the simulation of spiking neural networks (SNNs) and is geared towards machine learning and reinforcement learning.

BindsNET takes advantage of the `torch.Tensor` object to build spiking neurons and connections between them, and simulate them on CPUs or GPUs (for strong acceleration / parallelization) without any extra work. Recently, `torchvision.datasets` has been integrated into the library to allow the use of popular vision datasets in training SNNs for computer vision tasks. Neural network functionality contained in `torch.nn.functional` module is used to implement more complex connections between populations of spiking neurons.

Spiking neural networks are sometimes referred to as the third generation of neural networks. Rather than the simple linear layers and nonlinear activation functions of deep learning neural networks, SNNs are composed of neural units which more accurately capture properties of their biological counterparts. An important difference between spiking neurons and the artificial neurons of deep learning are the former's integration of input *in time*; they are naturally short-term memory devices by their maintenance of a (possibly decaying) membrane voltage. As a result, some have argued that SNNs are particularly well-suited to model time-varying data.

Neurons are connected together with directed edges (*synapses*) which are (in general) plastic. Synapses may have their own dynamics as well, which may or may not *depend on pre- and post-synaptic neural activity https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3395004/* or *other biological signals https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4717313/.* The modification of synaptic strengths is thought to be an important mechanism by which organisms learn. Accordingly, BindsNET provides a module (**bindsnet.learning**) which contains functions used for the updating of synapse weights.

At its core, BindsNET provides software objects and methods which support the simulation of groups of different types of neurons (**bindsnet.network.nodes**), as well as different types of connections between them (**bindsnet.network.topology**). These may be arbitrarily combined together under a single **bindsnet.network.Network** object, which is responsible for the coordination of the simulation logic of all underlying components. On creation of a network, the user can specify a simulation timestep constant, dt, which determines the granularity of the simulation. Choosing this parameter induces a trade-off between simulation speed and numerical precision: large values result in fast simulation, but poor simulation accuracy, and vice versa. Monitors (**bindsnet.network.monitors**) are available for recording state variables from arbitrary network components (e.g., the voltage v of a group of neurons).

The development of BindsNET is supported by the Defense Advanced Research Project Agency Grant DARPA/MTO HR0011-16-l-0006.

CHAPTER 1

Installation

1.1 Pip install

Issue:

```
pip install git+https://github.com/BindsNET/bindsnet.git
```

1.2 Installing from source

On *nix systems, issue one of the following in a shell:

```
git clone https://github.com/Hananel-Hazan/bindsnet.git  # HTTPS
git clone git@github.com:Hananel-Hazan/bindsnet.git  # SSH
```

Change directory into `bindsnet` and issue one of the following:

```
pip install . # Typical install
pip install -e .  # Editable mode (package code can be edited without reinstall)
```

This will install `bindsnet` and all its dependencies.

1.3 Running the tests

If BindsNET is installed from source, install `pytest` and issue the following from BindsNET's installation directory:

```
python -m pytest test
```

CHAPTER 2

Quickstart

Check out some example use cases for BindsNET in the `examples/` folder (link). For example, changing directory to *[bindsnet-root]/examples/mnist* and running the following will result in a near-replication of the architecture of Diehl & Cook 2015:

```
python eth_mnist.py [options]
```

The token `[options]` should be replaced with any command-line arguments you'd like to use to modify the behavior of the program.

CHAPTER 3

BindsNET User Manual

Welcome to BindsNET's user manual! To get started, click on one of the links below.

3.1 Part I: Creating and Adding Network Components

3.1.1 Creating a Network

The *bindsnet.network.Network* object is BindsNET's main offering. It is responsible for the coordination of simulation of all its constituent components: neurons, synapses, learning rules, etc. To create one:

```
from bindsnet.network import Network

network = Network()
```

The *bindsnet.network.Network* object accepts optional keyword arguments `dt`, `batch_size`, `learning`, and `reward_fn`.

The `dt` argument specifies the simulation time step, which determines what temporal granularity, in milliseconds, simulations are solved at. All simulation is done with the Euler method for the sake of computational simplicity. If instability is encountered in your simulation, use a smaller `dt` to resolve numerical instability.

The `batch_size` argument specifies the expected minibatch size of the input data. However, since BindsNET supports dynamics minibatch size, this argument can safely be ignored. It is used to initialize stateful neuronal and synaptic variables, and may provide a small speedup if specified beforehand.

The `learning` argument acts to enable or disable updates to adaptive parameters of network components; e.g., synapse weights or adaptive voltage thresholds. See **'Using Learning Rules'_** for more details.

The `reward_fn` argument takes in class that specifies how a scalar reward signal will be computed and fed to the network and its components. Typically, the output of this callable class will be used in certain "reward-modulated", or "three-factor" learning rules. See **'Using Learning Rules'_** for more details.

3.1.2 Adding Network Components

BindsNET supports three categories of network component: *layers* of neurons (`nodes`), *connections* between them (*bindsnet.network.topology*), and *monitors* for recording the evolution of state variables (*bindsnet.network.monitors*).

Note: Names of components in a network are arbitrary, and need only be unique within their component group (`layers`, `connections`, and `monitors`) in order to address them uniquely. We encourage our users to develop their own naming conventions, using whatever works best for them.

Creating and adding layers

To create a layer (or *population*) of nodes (in this case, leaky integrate-and-fire (LIF) neurons:

```
from bindsnet.network.nodes import LIFNodes

# Create a layer of 100 LIF neurons with shape (10, 10).
layer = LIFNodes(n=100, shape=(10, 10))
```

Each *bindsnet.network.nodes* object has many keyword arguments, but one of either `n` (the number of nodes in the layer, or `shape` (the arrangement of the layer, from which the number of nodes can be computed) is required. Other arguments for certain nodes objects include `thresh` (scalar or tensor giving voltage threshold(s) for the layer), `rest` (scalar or tensor giving resting voltage(s) for the layer), `traces` (whether to keep track of "spike traces" for each neuron in the layer), and `tc_decay` (scalar or tensor giving time constant(s) of the layer's neurons' voltage decay).

To add a layer to the network, use the `add_layer` function, and give it a name (a string) to call it by:

```
network.add_layer(
    layer=layer, name="LIF population"
)
```

Such layers are kept in the dictionary attribute `network.layers`, and can be accessed by the user; e.g., by `network.layers['LIF population']`.

Other layer types include *bindsnet.network.nodes.Input* (for user-specified input spikes), *bindsnet.network.nodes.McCullochPitts* (the McCulloch-Pitts neuron model), *bindsnet.network.nodes.AdaptiveLIFNodes* (LIF neurons with adaptive thresholds), and *bindsnet.network.nodes.IzhikevichNodes* (the Izhikevich neuron model). Any number of layers can be added to the network.

Custom nodes objects can be implemented by sub-classing *bindsnet.network.nodes.Nodes*, an abstract class with common logic for neuron simulation. The functions `forward(self, x: torch.Tensor)` (computes effects of input data on neuron population; e.g., voltage changes, spike occurrences, etc.), `reset_state_variables(self)` (resets neuron state variables to default values), and `_compute_decays(self)` must be implemented, as they are included as abstract functions of *bindsnet.network.nodes.Nodes*.

Creating and adding connections

Connections can be added between different populations of neurons (a *projection*), or from a population back to itself (a *recurrent* connection). To create an all-to-all connection:

```
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import Connection

# Create two populations of neurons, one to act as the "source"
# population, and the other, the "target population".
source_layer = Input(n=100)
target_layer = LIFNodes(n=1000)

# Connect the two layers.
connection = Connection(
    source=source_layer, target=target_layer
)
```

Like nodes, each connection object has many keyword arguments, but both `source` and `target` are required. These must be objects that subclass *bindsnet.network.nodes.Nodes*. Other arguments include `w` and `b` (weight and bias tensors for the connection), `wmin` and `wmax` (minimum and maximum allowable weight values), `update_rule` (*`bindsnet.learning.LearningRule`*; used for updating connection weights based on pre- and post-synaptic neuron activity and / or global neuromodulatory signals), and `norm` (a floating point value to normalize weights by).

To add a connection to the network, use the `add_connection` function, and pass the names given to source and target populations as `source` and `target` arguments. Make sure that the source and target neurons are added to the network as well:

```
network.add_layer(
    layer=source_layer, name="A"
)
network.add_layer(
    layer=target_layer, name="B"
)
network.add_connection(
    connection=connection, source="A", target="B"
)
```

Connections are kept in the dictionary attribute `network.connections`, and can be accessed by the user; e.g., by `network.connections['A', 'B']`. The layers must be added to the network with matching names (respectively, `A` and `B`) in order for the connection to work properly. There are no restrictions on the directionality of connections; layer "A" may connect to layer "B", and "B" back to "A", or "A" may connect directly back to itself.

Custom connection objects can be implemented by sub-classing *`bindsnet.network.topology.AbstractConnection`*, an abstract class with common logic for computing synapse outputs and updates. This includes functions `compute` (for computing input to downstream layer as a function of spikes and connection weights), `update` (for updating connection weights based on pre-, post-synaptic activity and possibly other signals; e.g., reward prediction error), `normalize` (for ensuring weights incident to post-synaptic neurons sum to a pre-specified value), and `reset_state_variables` (for re-initializing stateful variables for the start of a new simulation).

Specifying monitors

`bindsnet.network.monitors.AbstractMonitor` objects can be used to record tensor-valued variables over the course of simulation in certain network components. To create a monitor to monitor a single component:

```
from bindsnet.network import Network
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import Connection
from bindsnet.network.monitors import Monitor
```

(continues on next page)

(continued from previous page)

```
network = Network()

source_layer = Input(n=100)
target_layer = LIFNodes(n=1000)

connection = Connection(
    source=source_layer, target=target_layer
)

# Create a monitor.
monitor = Monitor(
    obj=target_layer,
    state_vars=("s", "v"),  # Record spikes and voltages.
    time=500,  # Length of simulation (if known ahead of time).
)
```

The user must specify a `Nodes` or `AbstractConnection` object from which to record, attributes of that object to record (`state_vars`), and, optionally, how many time steps the simulation(s) will last, in order to save time by pre-allocating memory.

To add a monitor to the network (thereby enabling monitoring), use the `add_monitor` function of the *bindsnet.network.Network* class:

```
network.add_layer(
    layer=source_layer, name="A"
)
network.add_layer(
    layer=target_layer, name="B"
)
network.add_connection(
    connection=connection, source="A", target="B"
)
network.add_monitor(monitor=monitor, name="B")
```

The name given to the monitor is not important. It is simply used by the user to select from the monitor objects controlled by a `Network` instance.

One can get the contents of a monitor by calling `network.monitors[<name>].get(<state_var>)`, where `<state_var>` is a member of the iterable passed in for the `state_vars` argument. This returns a tensor of shape `(time, n_1, ..., n_k)`, where `(n_1, ..., n_k)` is the shape of the recorded state variable.

The *bindsnet.network.monitors.NetworkMonitor* is used to record from many network components at once. To create one:

```
from bindsnet.network.monitors import NetworkMonitor

network_monitor = NetworkMonitor(
    network: Network,
    layers: Optional[Iterable[str]],
    connections: Optional[Iterable[Tuple[str, str]]],
    state_vars: Optional[Iterable[str]],
    time: Optional[int],
)
```

The user must specify the network to record from, an iterable of names of layers (entries in `network.layers`), an iterable of 2-tuples referring to connections (entries in `network.connections`), an iterable of tensor-valued state

variables to record during simulation (`state_vars`), and, optionally, how many time steps the simulation(s) will last, in order to save time by pre-allocating memory.

Similarly, one can get the contents of a network monitor by calling `network.monitors[<name>].get()`. Note this function takes no arguments; it returns a dictionary mapping network components to a sub-dictionary mapping state variables to their tensor-valued recording.

3.1.3 Running Simulations

After building up a `Network` object, the next step is to run a simulation. Here, the function `Network.run` comes into play. It takes arguments `inputs` (a dictionary mapping names of layers subclassing `AbstractInput` to input data of shape `[time, batch_size, *input_shape]`, where `input_shape` is the shape of the neuron population to which the data is passed), `time` (the number of simulation timesteps, generally thought of as milliseconds), and a number of keyword arguments, including `clamp` (and `unclamp`), used to force neurons to spike (or not spike) at any given time step, `reward`, for supplying to reward-modulated learning rules, and `masks`, a dictionary mapping connections to boolean tensors specifying which synapses weights to clamp to zero.

Building on the previous parts of this guide, we present a simple end-to-end example of simulating a two-layer, input-output spiking neural network.

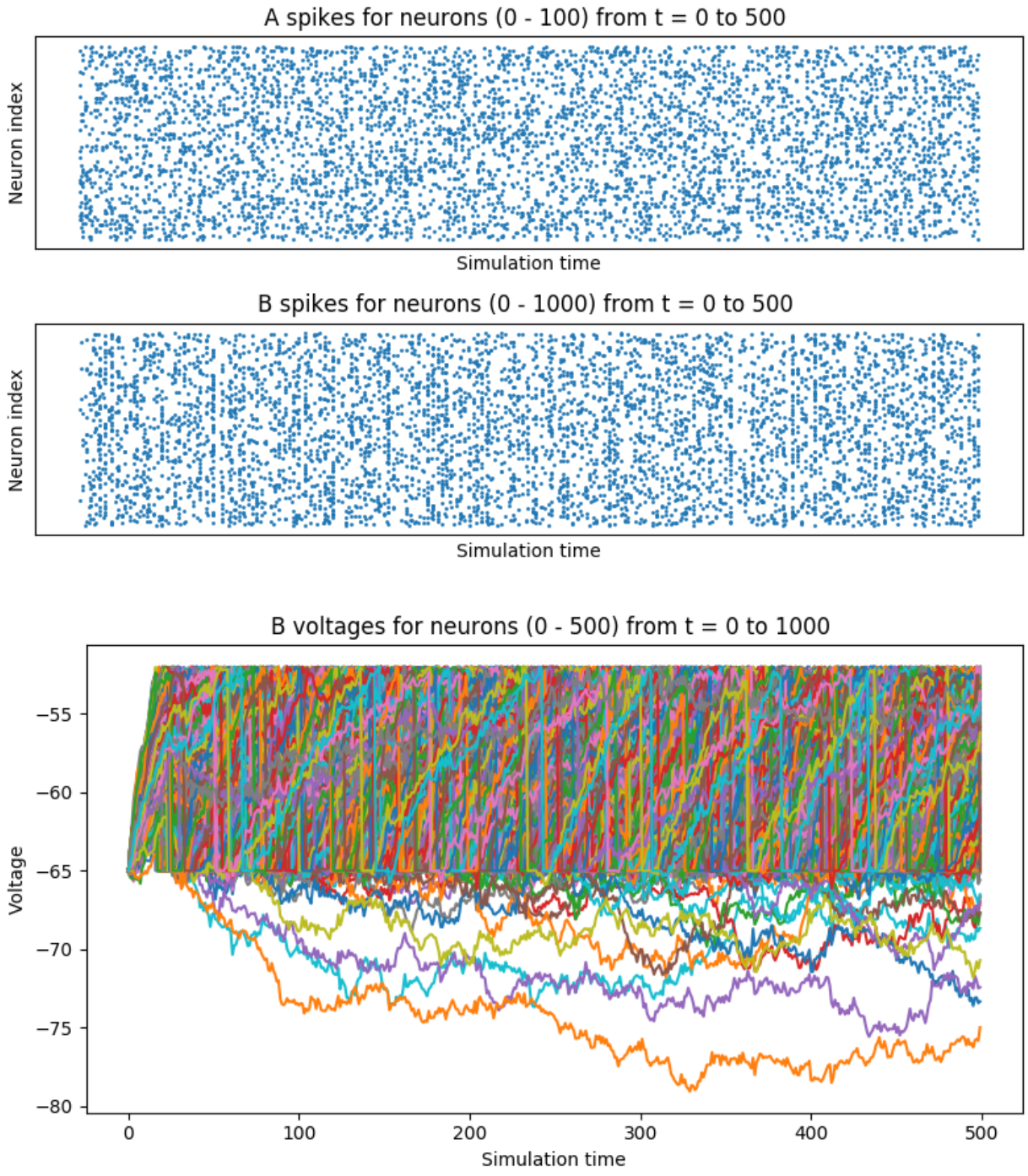
```
import torch
import matplotlib.pyplot as plt
from bindsnet.network import Network
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import Connection
from bindsnet.network.monitors import Monitor
from bindsnet.analysis.plotting import plot_spikes, plot_voltages

# Simulation time.
time = 500

# Create the network.
network = Network()

# Create and add input, output layers.
source_layer = Input(n=100)
target_layer = LIFNodes(n=1000)

network.add_layer(
    layer=source_layer, name="A"
)
network.add_layer(
    layer=target_layer, name="B"
)

# Create connection between input and output layers.
forward_connection = Connection(
    source=source_layer,
    target=target_layer,
    w=0.05 + 0.1 * torch.randn(source_layer.n, target_layer.n),  # Normal(0.05, 0.01)␣
↪weights.
)

network.add_connection(
    connection=forward_connection, source="A", target="B"
)
```

(continues on next page)

(continued from previous page)

```
# Create recurrent connection in output layer.
recurrent_connection = Connection(
    source=target_layer,
    target=target_layer,
    w=0.025 * (torch.eye(target_layer.n) - 1), # Small, inhibitory "competitive"␣
↪weights.
)

network.add_connection(
    connection=recurrent_connection, source="B", target="B"
)

# Create and add input and output layer monitors.
source_monitor = Monitor(
    obj=source_layer,
    state_vars=("s",),  # Record spikes and voltages.
    time=time,  # Length of simulation (if known ahead of time).
)
target_monitor = Monitor(
    obj=target_layer,
    state_vars=("s", "v"),  # Record spikes and voltages.
    time=time,  # Length of simulation (if known ahead of time).
)

network.add_monitor(monitor=source_monitor, name="A")
network.add_monitor(monitor=target_monitor, name="B")

# Create input spike data, where each spike is distributed according to Bernoulli(0.
↪1).
input_data = torch.bernoulli(0.1 * torch.ones(time, source_layer.n)).byte()
inputs = {"A": input_data}

# Simulate network on input data.
network.run(inputs=inputs, time=time)

# Retrieve and plot simulation spike, voltage data from monitors.
spikes = {
    "A": source_monitor.get("s"), "B": target_monitor.get("s")
}
voltages = {"B": target_monitor.get("v")}

plt.ioff()
plot_spikes(spikes)
plot_voltages(voltages, plot_type="line")
plt.show()
```

This script will result in figures that looks something like this:



Notice that, in the voltages plot, no voltage goes above -52mV, the default threshold of the `LIFNodes` object. After hitting this point, neurons' voltage is reset to -64mV, which can also be seen in the figure.

3.1.4 Simulation Notes

The simulation of all network components is *synchronous* (*clock-driven*); i.e., all components are updated at each time step. Other frameworks use *event-driven* simulation, where spikes can occur at arbitrary times instead of at regular

multiples of `dt`. We chose clock-driven simulation due to ease of implementation and for computational efficiency considerations.

During a simulation step, input to each layer is computed as the sum of all outputs from layers connecting to it (weighted by synapse weights) from the *previous* simulation time step (implemented by the `_get_inputs` method of the *`bindsnet.network.Network`* class). This model allows us to decouple network components and perform their simulation separately at the temporal granularity of chosen `dt`, interacting only between simulation steps.

This is a strict departure from the computation of *deep neural networks* (DNNs), in which an ordering of layers is supposed, and layers' activations are computed *in sequence* from the shallowest to the deepest layer in a single time step, with the exclusion of recurrent layers, whose computations are still ordered in time.

3.2 Part II: Creating and Adding Learning Rules

3.2.1 What is considered a learning rule?

Learning rules are necessary for the automated adaption of network parameters during simulation. At present, BindsNET supports two different categories of learning rules:

- **Two factor: Associative learning takes place based on pre- and post-synaptic neural activity. Examples include:**
  - The typical example is Hebbian learning, which may be summarized as "Cells that fire together wire together." That is, co-active neurons causes their connection strength to increase.
  - Spike-timing-dependent plasticity (STDP) stipulates that the ordering of pre- and post-synaptic spikes matters. A synapse is strengthened if the pre-synaptic neuron fires *before* the post-synaptic neuron, and, conversely, is weakened if it fires *after* the post-synaptic neuron. The magnitude of these updates is a decreasing function of the time between pre- and post-synaptic spikes.
- **Three factor: In addition to associating pre- and post-synaptic neural activity, a third factor is introduced which modulate**
  - (Reward, error, attention)-modulated (Hebbian learning, STDP): The same learning rules described above are modulated by the presence of global signals such as reward, error, or attention, which can be variously defined in machine learning or reinforcement learning contexts. These signals act to gate plasticity, turning it on or off and switching its sign and magnitude, based on the task at hand.

The above are examples of local learning rules, where the information needed to make updates are thought to be available at the synapse. For example, pre- and post-synaptic neurons are adjacent to synapses, rendering their spiking activity accessible, whereas chemical signals like dopamine (hypothesized to be a reward prediction error (RPE) signal) are widely distributed across certain neuron populations; i.e., they are *globally* available. This is in contrast to learning algorithms like back-propagation, where per-synapse error signals are derived by computing backwards from a loss function at the network's output layer. Such error derivation is thought to be biologically implausible, especially compared to the two- and three-factor rules mentioned above.

3.2.2 Creating a learning rule in BindsNET

At present, learning rules are attached to specific `Connection` objects. For example, to create a connection with a STDP learning rule on the synapses:

```
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import Connection
from bindsnet.learning import PostPre

```

(continues on next page)

(continued from previous page)

```
# Create two populations of neurons, one to act as the "source"
# population, and the other, the "target population".
# Neurons involved in certain learning rules must record synaptic
# traces, a vector of short-term memories of the last emitted spikes.
source_layer = Input(n=100, traces=True)
target_layer = LIFNodes(n=1000, traces=True)

# Connect the two layers.
connection = Connection(
    source=source_layer, target=target_layer, update_rule=PostPre, nu=(1e-4, 1e-2)
)
```

The connection may be added to a `Network` instance as usual. The `Connection` object takes arguments `update_rule`, of type `bindsnet.learning.LearningRule`, as well as `nu`, a 2-tuple specifying pre- and post-synaptic learning rates; i.e., multiplicative factors which modulate how quickly synapse weights change.

Learning rules also accept arguments `reduction`, which specifies how parameter updates are aggregated across the batch dimension, and `weight_decay`, which specifies the time constant of the rate of decay of synapse weights to zero. By default, parameter updates are averaged across the batch dimension, and there is no weight decay.

Other supported learning rules include `Hebbian`, `WeightDependentPostPre`, `MSTDP` (reward-modulated STDP), and `MSTDPET` (reward-modulated STDP with eligibility traces).

Custom learning rules can be implemented by subclassing `bindsnet.learning.LearningRule` and providing implementations for the types of `AbstractConnection` objects intended to be used. For example, the `Connection` and `LocalConnection` objects rely on the implementation of a private method, `_connection_update`, whereas the `Conv2dConnection` object uses the `_conv2d_connection_update` version.

CHAPTER 4

bindsnet package

4.1 Subpackages

4.1.1 bindsnet.analysis package

Submodules

bindsnet.analysis.pipeline_analysis module

class bindsnet.analysis.pipeline_analysis.**MatplotlibAnalyzer**(***kwargs*)
Bases: *bindsnet.analysis.pipeline_analysis.PipelineAnalyzer*

Renders output using Matplotlib.

Matplotlib requires objects to be kept around over the full lifetime of the plots; this is done through `self.plots`. An interactive session is needed so that we can continue processing and just update the plots.

Initializes the analyzer.

Keyword arguments:

Parameters **volts_type** (*str*) – Type of plotting for voltages (`"color"` or `"line"`).

finalize_step() → None
Flush the output from the current step

plot_conv2d_weights(*weights: torch.Tensor*, *tag: str = 'conv2d'*, *step: int = None*) → None
Plot a connection weight matrix of a `Conv2dConnection`.

Parameters

- **weights** – Weight matrix of `Conv2dConnection` object.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_obs(*obs: torch.Tensor*, *tag: str = 'obs'*, *step: int = None*) → None
Pulls the observation off of torch and sets up for Matplotlib plotting.

Parameters

- **obs** – A 2D array of floats depicting an input image.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_reward(*reward_list: list*, *reward_window: int = None*, *tag: str = 'reward'*, *step: int = None*) → None
Plot the accumulated reward for each episode.

Parameters

- **reward_list** – The list of recent rewards to be plotted.
- **reward_window** – The length of the window to compute a moving average over.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_spikes(*spike_record: Dict[str, torch.Tensor], tag: str = 'spike', step: int = None*) → None
Plots all spike records inside of `spike_record`. Keeps unique plots for all unique tags that are given.

Parameters

- **spike_record** – Dictionary of spikes to be rasterized.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_voltages(*voltage_record: Dict[str, torch.Tensor], thresholds: Optional[Dict[str, torch.Tensor]] = None, tag: str = 'voltage', step: int = None*) → None
Plots all voltage records and given thresholds. Keeps unique plots for all unique tags that are given.

Parameters

- **voltage_record** – Dictionary of voltages for neurons inside of networks organized by the layer they correspond to.
- **thresholds** – Optional dictionary of threshold values for neurons.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

class bindsnet.analysis.pipeline_analysis.**PipelineAnalyzer**
Bases: abc.ABC

Responsible for pipeline analysis. Subclasses maintain state information related to plotting or logging.

finalize_step() → None
Flush the output from the current step.

plot_conv2d_weights(*weights: torch.Tensor*, *tag: str = 'conv2d'*, *step: int = None*) → None
Plot a connection weight matrix of a `Conv2dConnection`.

Parameters

- **weights** – Weight matrix of `Conv2dConnection` object.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_obs(*obs: torch.Tensor*, *tag: str = 'obs'*, *step: int = None*) → None
Pulls the observation from PyTorch and sets up for Matplotlib plotting.

Parameters

- **obs** – A 2D array of floats depicting an input image.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_reward(*reward_list: list*, *reward_window: int = None*, *tag: str = 'reward'*, *step: int = None*) → None
Plot the accumulated reward for each episode.

Parameters

- **reward_list** – The list of recent rewards to be plotted.
- **reward_window** – The length of the window to compute a moving average over.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_spikes(*spike_record: Dict[str, torch.Tensor], tag: str = 'spike', step: int = None*) → None
Plots all spike records inside of `spike_record`. Keeps unique plots for all unique tags that are given.

Parameters

- **spike_record** – Dictionary of spikes to be rasterized.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_voltages(*voltage_record: Dict[str, torch.Tensor], thresholds: Optional[Dict[str, torch.Tensor]] = None, tag: str = 'voltage', step: int = None*) → None
Plots all voltage records and given thresholds. Keeps unique plots for all unique tags that are given.

Parameters

- **voltage_record** – Dictionary of voltages for neurons inside of networks organized by the layer they correspond to.
- **thresholds** – Optional dictionary of threshold values for neurons.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

class bindsnet.analysis.pipeline_analysis.**TensorboardAnalyzer**(*summary_directory: str = './logs'*)
Bases: *bindsnet.analysis.pipeline_analysis.PipelineAnalyzer*

Initializes the analyzer.

Parameters **summary_directory** – Directory to save log files.

finalize_step() → None
No-op for `TensorboardAnalyzer`.

plot_conv2d_weights(*weights: torch.Tensor*, *tag: str = 'conv2d'*, *step: int = None*) → None
Plot a connection weight matrix of a `Conv2dConnection`.

Parameters

- **weights** – Weight matrix of `Conv2dConnection` object.

- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_obs(*obs: torch.Tensor*, *tag: str = 'obs'*, *step: int = None*) → None
Pulls the observation off of torch and sets up for Matplotlib plotting.

Parameters

- **obs** – A 2D array of floats depicting an input image.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_reward(*reward_list: list*, *reward_window: int = None*, *tag: str = 'reward'*, *step: int = None*) → None
Plot the accumulated reward for each episode.

Parameters

- **reward_list** – The list of recent rewards to be plotted.
- **reward_window** – The length of the window to compute a moving average over.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_spikes(*spike_record: Dict[str, torch.Tensor], tag: str = 'spike', step: int = None*) → None
Plots all spike records inside of `spike_record`. Keeps unique plots for all unique tags that are given.

Parameters

- **spike_record** – Dictionary of spikes to be rasterized.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

plot_voltages(*voltage_record: Dict[str, torch.Tensor], thresholds: Optional[Dict[str, torch.Tensor]] = None, tag: str = 'voltage', step: int = None*) → None
Plots all voltage records and given thresholds. Keeps unique plots for all unique tags that are given.

Parameters

- **voltage_record** – Dictionary of voltages for neurons inside of networks organized by the layer they correspond to.
- **thresholds** – Optional dictionary of threshold values for neurons.
- **tag** – A unique tag to associate the data with.
- **step** – The step of the pipeline.

bindsnet.analysis.plotting module

bindsnet.analysis.plotting.**plot_assignments**(*assignments: torch.Tensor*, *im: Optional[matplotlib.image.AxesImage] = None*, *figsize: Tuple[int, int] = (5, 5)*, *classes: Optional[Sized] = None*, *save: Optional[str] = None*) → matplotlib.image.AxesImage
Plot the two-dimensional neuron assignments.

Parameters

- **assignments** – Vector of neuron label assignments.
- **im** – Used for re-drawing the assignments plot.
- **figsize** – Horizontal, vertical figure size in inches.
- **classes** – Iterable of labels for colorbar ticks corresponding to data labels.
- **save** – file name to save fig, if None = not saving fig.

Returns Used for re-drawing the assigments plot.

bindsnet.analysis.plotting.**plot_conv2d_weights**(*weights: torch.Tensor*, *wmin: float = 0.0*, *wmax: float = 1.0*, *im: Optional[matplotlib.image.AxesImage] = None*, *figsize: Tuple[int, int] = (5, 5)*, *cmap: str = 'hot_r'*) → matplotlib.image.AxesImage

Plot a connection weight matrix of a Conv2dConnection.

Parameters

- **weights** – Weight matrix of Conv2dConnection object.
- **wmin** – Minimum allowed weight value.
- **wmax** – Maximum allowed weight value.
- **im** – Used for re-drawing the weights plot.
- **figsize** – Horizontal, vertical figure size in inches.
- **cmap** – Matplotlib colormap.

Returns Used for re-drawing the weights plot.

bindsnet.analysis.plotting.**plot_input**(*image: torch.Tensor*, *inpt: torch.Tensor*, *label: Optional[int] = None*, *axes: List[matplotlib.axes._axes.Axes] = None*, *ims: List[matplotlib.image.AxesImage] = None*, *figsize: Tuple[int, int] = (8, 4)*) → Tuple[List[matplotlib.axes._axes.Axes], List[matplotlib.image.AxesImage]]

Plots a two-dimensional image and its corresponding spike-train representation.

Parameters

- **image** – A 2D array of floats depicting an input image.
- **inpt** – A 2D array of floats depicting an image's spike-train encoding.
- **label** – Class label of the input data.
- **axes** – Used for re-drawing the input plots.
- **ims** – Used for re-drawing the input plots.
- **figsize** – Horizontal, vertical figure size in inches.

Returns Tuple of `(axes, ims)` used for re-drawing the input plots.

bindsnet.analysis.plotting.**plot_local_connection_2d_weights**(*lc: object, input_channel: int = 0, output_channel: int = None, im: Optional[matplotlib.image.AxesImage] = None, lines: bool = True, figsize: Tuple[int, int] = (5, 5), cmap: str = 'hot_r', color: str = 'r'*) → matplotlib.image.AxesImage

Plot a connection weight matrix of a `Connection` with *locally connected structure <http://yann.lecun.com/exdb/publis/pdf/gregor-nips-11.pdf>_. :param lc: An object of the class LocalConnection2D :param input_channel: The input channel to plot its corresponding weights, default is the first channel :param output_channel: If not None, will only plot the weights corresponding to this output channel (filter) :param lines: Indicates whether or not draw horizontal and vertical lines separating input regions. :param figsize: Horizontal and vertical figure size in inches. :param cmap: Matplotlib colormap. :return: ``ims, axes``*: Used for re-drawing the plots.

bindsnet.analysis.plotting.**plot_locally_connected_weights**(*weights: torch.Tensor, n_filters: int, kernel_size: Union[int, Tuple[int, int]], conv_size: Union[int, Tuple[int, int]], locations: torch.Tensor, input_sqrt: Union[int, Tuple[int, int]], wmin: float = 0.0, wmax: float = 1.0, im: Optional[matplotlib.image.AxesImage] = None, lines: bool = True, figsize: Tuple[int, int] = (5, 5), cmap: str = 'hot_r'*) → matplotlib.image.AxesImage

Plot a connection weight matrix of a `Connection` with ‘locally connected structure <http://yann.lecun.com/exdb/publis/pdf/gregor-nips-11.pdf>_.

Parameters

- **weights** – Weight matrix of Conv2dConnection object.
- **n_filters** – No. of convolution kernels in use.
- **kernel_size** – Side length(s) of 2D convolution kernels.
- **conv_size** – Side length(s) of 2D convolution population.
- **locations** – Indices of input receptive fields for convolution population neurons.
- **input_sqrt** – Side length(s) of 2D input data.
- **wmin** – Minimum allowed weight value.
- **wmax** – Maximum allowed weight value.
- **im** – Used for re-drawing the weights plot.

- **lines** – Whether or not to draw horizontal and vertical lines separating input regions.
- **figsize** – Horizontal, vertical figure size in inches.
- **cmap** – Matplotlib colormap.

Returns Used for re-drawing the weights plot.

bindsnet.analysis.plotting.**plot_performance**(*performances: Dict[str, List[float]], ax: Optional[matplotlib.axes._axes.Axes] = None, figsize: Tuple[int, int] = (7, 4), x_scale: int = 1, save: Optional[str] = None*) → matplotlib.axes._axes.Axes

Plot training accuracy curves.

Parameters

- **performances** – Lists of training accuracy estimates per voting scheme.
- **ax** – Used for re-drawing the performance plot.
- **figsize** – Horizontal, vertical figure size in inches.
- **x_scale** – scaling factor for the x axis, equal to the number of examples per performance measure
- **save** – file name to save fig, if None = not saving fig.

Returns Used for re-drawing the performance plot.

bindsnet.analysis.plotting.**plot_spikes**(*spikes: Dict[str, torch.Tensor], time: Optional[Tuple[int, int]] = None, n_neurons: Optional[Dict[str, Tuple[int, int]]] = None, ims: Optional[List[matplotlib.collections.PathCollection]] = None, axes: Union[matplotlib.axes._axes.Axes, List[matplotlib.axes._axes.Axes], None] = None, figsize: Tuple[float, float] = (8.0, 4.5)*) → Tuple[List[matplotlib.image.AxesImage], List[matplotlib.axes._axes.Axes]]

Plot spikes for any group(s) of neurons.

Parameters

- **spikes** – Mapping from layer names to spiking data. Spike data has shape `[time, n_1, ..., n_k]`, where `[n_1, ..., n_k]` is the shape of the recorded layer.
- **time** – Plot spiking activity of neurons in the given time range. Default is entire simulation time.
- **n_neurons** – Plot spiking activity of neurons in the given range of neurons. Default is all neurons.
- **ims** – Used for re-drawing the plots.
- **axes** – Used for re-drawing the plots.
- **figsize** – Horizontal, vertical figure size in inches.

Returns `ims, axes`: Used for re-drawing the plots.

bindsnet.analysis.plotting.**plot_voltages**(*voltages: Dict[str, torch.Tensor], ims: Optional[List[matplotlib.image.AxesImage]] = None, axes: Optional[List[matplotlib.axes._axes.Axes]] = None, time: Tuple[int, int] = None, n_neurons: Optional[Dict[str, Tuple[int, int]]] = None, cmap: Optional[str] = 'jet', plot_type: str = 'color', thresholds: Dict[str, torch.Tensor] = None, figsize: Tuple[float, float] = (8.0, 4.5)*) → Tuple[List[matplotlib.image.AxesImage], List[matplotlib.axes._axes.Axes]]

Plot voltages for any group(s) of neurons.

Parameters

- **voltages** – Contains voltage data by neuron layers.
- **ims** – Used for re-drawing the plots.
- **axes** – Used for re-drawing the plots.
- **time** – Plot voltages of neurons in given time range. Default is entire simulation time.
- **n_neurons** – Plot voltages of neurons in given range of neurons. Default is all neurons.
- **cmap** – Matplotlib colormap to use.
- **figsize** – Horizontal, vertical figure size in inches.
- **plot_type** – The way how to draw graph. 'color' for pcolormesh, 'line' for curved lines.
- **thresholds** – Thresholds of the neurons in each layer.

Returns `ims, axes`: Used for re-drawing the plots.

bindsnet.analysis.plotting.**plot_weights**(*weights: torch.Tensor*, *wmin: Optional[float] = 0*, *wmax: Optional[float] = 1*, *im: Optional[matplotlib.image.AxesImage] = None*, *figsize: Tuple[int*, *int] = (5*, *5)*, *cmap: str = 'hot_r'*, *save: Optional[str] = None*) → matplotlib.image.AxesImage

Plot a connection weight matrix.

Parameters

- **weights** – Weight matrix of `Connection` object.
- **wmin** – Minimum allowed weight value.
- **wmax** – Maximum allowed weight value.
- **im** – Used for re-drawing the weights plot.
- **figsize** – Horizontal, vertical figure size in inches.
- **cmap** – Matplotlib colormap.
- **save** – file name to save fig, if None = not saving fig.

Returns `AxesImage` for re-drawing the weights plot.

bindsnet.analysis.visualization module

bindsnet.analysis.visualization.**plot_spike_trains_for_example**(*spikes: torch.Tensor*, *n_ex: Optional[int] = None*, *top_k: Optional[int] = None*, *indices: Optional[List[int]] = None*) → None

Plot spike trains for top-k neurons or for specific indices.

Parameters

- **spikes** – Spikes for one simulation run of shape `(n_examples, n_neurons, time)`.
- **n_ex** – Allows user to pick which example to plot spikes for.
- **top_k** – Plot k neurons that spiked the most for n_ex example.
- **indices** – Plot specific neurons' spiking activity instead of top_k.

bindsnet.analysis.visualization.**plot_voltage**(*voltage: torch.Tensor*, *n_ex: int = 0*, *n_neuron: int = 0*, *time: Optional[Tuple[int, int]] = None*, *threshold: float = None*) → None

Plot voltage for a single neuron on a specific example.

Parameters

- **voltage** – Tensor or array of shape `[n_examples, n_neurons, time]`.
- **n_ex** – Allows user to pick which example to plot voltage for.
- **n_neuron** – Neuron index for which to plot voltages for.
- **time** – Plot spiking activity of neurons between the given range of time.
- **threshold** – Neuron spiking threshold.

bindsnet.analysis.visualization.**plot_weights_movie**(*ws: numpy.ndarray*, *sample_every: int = 1*) → None

Create and plot movie of weights.

Parameters

- **ws** – Array of shape `[n_examples, source, target, time]`.
- **sample_every** – Sub-sample using this parameter.

bindsnet.analysis.visualization.**summary**(*net*) → str

Summarizes informations about a Network. Includes layers and connection informations.

Parameters **net** – Network

Returns string

Module contents

4.1.2 bindsnet.conversion package

Submodules

bindsnet.conversion.conversion module

class bindsnet.conversion.conversion.**FeatureExtractor**(*submodule*)
Bases: torch.nn.modules.module.Module

Special-purpose PyTorch module for the extraction of child module's activations.

Constructor for FeatureExtractor module.

Parameters **submodule** – The module who's children modules are to be extracted.

forward(*x: torch.Tensor*) → Dict[torch.nn.modules.module.Module, torch.Tensor]
Forward pass of the feature extractor.

Parameters **x** – Input data for the "submodule".

Returns A dictionary mapping

class bindsnet.conversion.conversion.**Permute**(*dims*)
Bases: torch.nn.modules.module.Module

PyTorch module for the explicit permutation of a tensor's dimensions in a parent module's forward pass (as opposed to torch.permute).

Constructor for Permute module.

Parameters **dims** – Ordering of dimensions for permutation.

forward(*x*)
Forward pass of permutation module.

Parameters **x** – Input tensor to permute.

Returns Permuted input tensor.

bindsnet.conversion.conversion.**ann_to_snn**(*ann: Union[torch.nn.modules.module.Module, str], input_shape: Sequence[int], data: Optional[torch.Tensor] = None, percentile: float = 99.9, node_type: Optional[bindsnet.network.nodes.Nodes] = <class 'bindsnet.conversion.nodes.SubtractiveResetIFNodes'>, **kwargs*) → bindsnet.network.network.Network
Converts an artificial neural network (ANN) written as a torch.nn.Module into a near-equivalent spiking neural network.

Parameters

- **ann** – Artificial neural network implemented in PyTorch. Accepts either torch.nn.Module or path to network saved using torch.save().
- **input_shape** – Shape of input data.
- **data** – Data to use to perform data-based weight normalization of shape [n_examples, ...].

- **percentile** – Percentile (in `[0, 100]`) of activations to scale by in data-based normalization scheme.
- **node_type** – Class of `Nodes` to use in replacing `torch.nn.Linear` layers in original ANN.

Returns Spiking neural network implemented in PyTorch.

`bindsnet.conversion.conversion.`**data_based_normalization**(*ann: Union[torch.nn.modules.module.Module, str], data: torch.Tensor, percentile: float = 99.9*)

Use a dataset to rescale ANN weights and biases such that that the max ReLU activation is less than 1.

Parameters

- **ann** – Artificial neural network implemented in PyTorch. Accepts either `torch.nn.Module` or path to network saved using `torch.save()`.
- **data** – Data to use to perform data-based weight normalization of shape `[n_examples, ...]`.
- **percentile** – Percentile (in `[0, 100]`) of activations to scale by in data-based normalization scheme.

Returns Artificial neural network with rescaled weights and biases according to activations on the dataset.

bindsnet.conversion.nodes module

class `bindsnet.conversion.nodes.`**PassThroughNodes**(*n: Optional[int] = None, shape: Optional[Iterable[int]] = None, traces: bool = False, traces_additive: bool = False, tc_trace: Union[float, torch.Tensor] = 20.0, trace_scale: Union[float, torch.Tensor] = 1.0, sum_input: bool = False*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of integrate-and-fire (IF) neurons with using reset by subtraction.

Instantiates a layer of IF neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **trace_tc** – Time constant of spike trace decay.
- **sum_input** – Whether to sum all inputs.

forward(*x: torch.Tensor*) → None

Runs a single simulation step.

Parameters

- **inputs** – Inputs to the layer.
- **dt** – Simulation time step.

reset_state_variables() → None
Resets relevant state variables.

class bindsnet.conversion.nodes.**SubtractiveResetIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of *integrate-and-fire (IF) neurons <https://bit.ly/2EOk6YN>* using reset by subtraction.

Instantiates a layer of IF neurons with the subtractive reset mechanism from this paper.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **lbound** – Lower bound of the voltage.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

bindsnet.conversion.topology module

class bindsnet.conversion.topology.**ConstantPad2dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, padding: Tuple, nu: Union[float, Iterable[float], None] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Special-purpose connection for emulating the `ConstantPad2d` PyTorch module in spiking neural networks.

Constructor for `ConstantPad2dConnection`.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **padding** – Padding of input tensors; passed to `torch.nn.functional.pad`.
- **nu** – Learning rate for both pre- and post-synaptic events.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*function*) – Modifies connection parameters according to some rule.
- **wmin** (*float*) – The minimum value on the connection weights.
- **wmax** (*float*) – The maximum value on the connection weights.
- **norm** (*float*) – Total weight per target neuron normalization.

compute(*s: torch.Tensor*)

Pad input.

Parameters **s** – Input.

Returns Padding input.

class bindsnet.conversion.topology.**PermuteConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, dims: Iterable[T_co], nu: Union[float, Iterable[float], None] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Special-purpose connection for emulating the custom `Permute` module in spiking neural networks.

Constructor for `PermuteConnection`.

Parameters

- **source** – A layer of nodes from which the connection originates.

- **target** – A layer of nodes to which the connection connects.
- **dims** – Order of dimensions to permute.
- **nu** – Learning rate for both pre- and post-synaptic events.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*function*) – Modifies connection parameters according to some rule.
- **wmin** (*float*) – The minimum value on the connection weights.
- **wmax** (*float*) – The maximum value on the connection weights.
- **norm** (*float*) – Total weight per target neuron normalization.

compute(*s: torch.Tensor*) → torch.Tensor
Permute input.

Parameters **s** – Input.

Returns Permuted input.

Module contents

class bindsnet.conversion.**Permute**(*dims*)
Bases: torch.nn.modules.module.Module

PyTorch module for the explicit permutation of a tensor's dimensions in a parent module's `forward` pass (as opposed to `torch.permute`).

Constructor for `Permute` module.

Parameters **dims** – Ordering of dimensions for permutation.

forward(*x*)
Forward pass of permutation module.

Parameters **x** – Input tensor to permute.

Returns Permuted input tensor.

class bindsnet.conversion.**FeatureExtractor**(*submodule*)
Bases: torch.nn.modules.module.Module

Special-purpose PyTorch module for the extraction of child module's activations.

Constructor for `FeatureExtractor` module.

Parameters **submodule** – The module who's children modules are to be extracted.

forward(*x: torch.Tensor*) → Dict[torch.nn.modules.module.Module, torch.Tensor]
Forward pass of the feature extractor.

Parameters **x** – Input data for the ‘‘submodule”.

Returns A dictionary mapping

class bindsnet.conversion.**SubtractiveResetIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of *integrate-and-fire (IF) neurons <https://bit.ly/2EOk6YN>* using reset by subtraction.

Instantiates a layer of IF neurons with the subtractive reset mechanism from this paper.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **lbound** – Lower bound of the voltage.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.conversion.**PassThroughNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of integrate-and-fire (IF) neurons with using reset by subtraction.

Instantiates a layer of IF neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **trace_tc** – Time constant of spike trace decay.
- **sum_input** – Whether to sum all inputs.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters

- **inputs** – Inputs to the layer.
- **dt** – Simulation time step.

reset_state_variables() → None
Resets relevant state variables.

class bindsnet.conversion.**PermuteConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, dims: Iterable[T_co], nu: Union[float, Iterable[float], None] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Special-purpose connection for emulating the custom `Permute` module in spiking neural networks.

Constructor for `PermuteConnection`.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **dims** – Order of dimensions to permute.
- **nu** – Learning rate for both pre- and post-synaptic events.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*function*) – Modifies connection parameters according to some rule.
- **wmin** (*float*) – The minimum value on the connection weights.
- **wmax** (*float*) – The maximum value on the connection weights.
- **norm** (*float*) – Total weight per target neuron normalization.

compute(*s: torch.Tensor*) → torch.Tensor
Permute input.

Parameters **s** – Input.

Returns Permuted input.

class bindsnet.conversion.**ConstantPad2dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, padding: Tuple, nu: Union[float, Iterable[float], None] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Special-purpose connection for emulating the `ConstantPad2d` PyTorch module in spiking neural networks.

Constructor for `ConstantPad2dConnection`.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **padding** – Padding of input tensors; passed to `torch.nn.functional.pad`.
- **nu** – Learning rate for both pre- and post-synaptic events.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*function*) – Modifies connection parameters according to some rule.
- **wmin** (*float*) – The minimum value on the connection weights.
- **wmax** (*float*) – The maximum value on the connection weights.
- **norm** (*float*) – Total weight per target neuron normalization.

compute(*s: torch.Tensor*)

Pad input.

Parameters **s** – Input.

Returns Padding input.

bindsnet.conversion.**data_based_normalization**(*ann: Union[torch.nn.modules.module.Module, str], data: torch.Tensor, percentile: float = 99.9*)

Use a dataset to rescale ANN weights and biases such that that the max ReLU activation is less than 1.

Parameters

- **ann** – Artificial neural network implemented in PyTorch. Accepts either `torch.nn.Module` or path to network saved using `torch.save()`.
- **data** – Data to use to perform data-based weight normalization of shape `[n_examples, ...]`.
- **percentile** – Percentile (in `[0, 100]`) of activations to scale by in data-based normalization scheme.

Returns Artificial neural network with rescaled weights and biases according to activations on the dataset.

bindsnet.conversion.**ann_to_snn**(*ann: Union[torch.nn.modules.module.Module, str], input_shape: Sequence[int], data: Optional[torch.Tensor] = None, percentile: float = 99.9, node_type: Optional[bindsnet.network.nodes.Nodes] = <class 'bindsnet.conversion.nodes.SubtractiveResetIFNodes'>, **kwargs*) → bindsnet.network.network.Network

Converts an artificial neural network (ANN) written as a `torch.nn.Module` into a near-equivalent spiking neural network.

Parameters

- **ann** – Artificial neural network implemented in PyTorch. Accepts either `torch.nn.Module` or path to network saved using `torch.save()`.
- **input_shape** – Shape of input data.
- **data** – Data to use to perform data-based weight normalization of shape `[n_examples, ...]`.
- **percentile** – Percentile (in `[0, 100]`) of activations to scale by in data-based normalization scheme.
- **node_type** – Class of `Nodes` to use in replacing `torch.nn.Linear` layers in original ANN.

Returns Spiking neural network implemented in PyTorch.

4.1.3 bindsnet.datasets package

Submodules

bindsnet.datasets.alov300 module

class bindsnet.datasets.alov300.**ALOV300**(*root*, *transform*, *input_size*, *download=False*)

Bases: `torch.utils.data.dataset.Dataset`

Class to read the ALOV dataset

Parameters

- **root** – Path to the ALOV folder that contains JPEGImages, annotations, etc. folders.
- **input_size** – The input size of network that is using this data, for rescaling.
- **download** – Specify whether to download the dataset if it is not present.
- **num_samples** – Number of samples to pass to the batch

DATASET_WEB = 'http://alov300pp.joomlafree.it/dataset-resources.html'

VOID_LABEL = 255

get_bb(*ann*)

Parses ALOV annotation and returns bounding box in the format: [left, upper, width, height]

get_orig_sample(*idx*, *i=1*)

Returns original image with bounding box at a specific index. Range of valid index: [0, self.len-1].

get_sample(*idx*)

Returns sample without transformation for visualization.

Sample consists of resized previous and current frame with target which is passed to the network. Bounding box values are normalized between 0 and 1 with respect to the target frame and then scaled by factor of 10.

progress(*count*, *block_size*, *total_size*)

show(*idx*, *is_current=1*)
Helper function to display image at a particular index with grounttruth bounding box.

Arguments: idx: index is_current: 0 for previous frame and 1 for current frame

show_sample(*idx*)
Helper function to display sample, which is passed to GOTURN. Shows previous frame and current frame with bounding box.

bindsnet.datasets.collate module

This code is directly pulled from the pytorch version found at:

https://github.com/pytorch/pytorch/blob/master/torch/utils/data/_utils/collate.py

Modifications exist to have [time, batch, n_0, ... n_k] instead of batch in dimension 0.

bindsnet.datasets.collate.**safe_worker_check**()
Method to check to use shared memory.

bindsnet.datasets.collate.**time_aware_collate**(*batch*)
Puts each data field into a tensor with dimensions `[time, batch size, ...]`

Interpretation of dimensions being input: - 0 dim (,) - (1, batch_size, 1) - 1 dim (time,) - (time, batch_size, 1) - >2 dim (time, n_0, ...) - (time, batch_size, n_0, ...)

bindsnet.datasets.dataloader module

class bindsnet.datasets.dataloader.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=<function time_aware_collate>*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*)
Bases: `torch.utils.data.dataloader.DataLoader`

bindsnet.datasets.davis module

class bindsnet.datasets.davis.**Davis**(*root*, *task='unsupervised'*, *subset='train'*, *sequences='all'*, *resolution='480p'*, *size=(600, 480)*, *codalab=False*, *download=False*, *num_samples: int = -1*)
Bases: `torch.utils.data.dataset.Dataset`

Class to read the DAVIS dataset :param root: Path to the DAVIS folder that contains JPEGImages, Annotations,

etc. folders.

Parameters

- **task** – Task to load the annotations, choose between semi-supervised or unsupervised.
- **subset** – Set to load the annotations

- **sequences** – Sequences to consider, 'all' to use all the sequences in a set.
- **resolution** – Specify the resolution to use the dataset, choose between '480' and 'Full-Resolution'
- **download** – Specify whether to download the dataset if it is not present
- **num_samples** – Number of samples to pass to the batch

DATASET_WEB = 'https://davischallenge.org/davis2017/code.html'

RESOLUTION_OPTIONS = ['480p', 'Full-Resolution']

SUBSET_OPTIONS = ['train', 'val', 'test-dev', 'test-challenge']

TASKS = ['semi-supervised', 'unsupervised']

VOID_LABEL = 255

get_all_images(*sequence*)

get_all_masks(*sequence*, *separate_objects_masks=False*)

get_frames(*sequence*)

get_sequences()

static progress(*count*, *block_size*, *total_size*)
Simple progress indicator for the download of the dataset.

bindsnet.datasets.preprocess module

class bindsnet.datasets.preprocess.**BoundingBox**(*x1*, *y1*, *x2*, *y2*)
Bases: object

compute_output_height()

compute_output_width()

edge_spacing_x()

edge_spacing_y()

get_bb_list()

get_center_x()

get_center_y()

get_height()

get_width()

print_bb()

recenter(*search_loc*, *edge_spacing_x*, *edge_spacing_y*, *bbox_gt_recentered*)

scale(*image*)

shift(*image*, *lambda_scale_frac*, *lambda_shift_frac*, *min_scale*, *max_scale*, *shift_motion_model*, *bbox_rand*)

uncenter(*raw_image*, *search_location*, *edge_spacing_x*, *edge_spacing_y*)

unscale(*image*)

class bindsnet.datasets.preprocess.**NormalizeToTensor**
Bases: object

Returns torch tensor normalized images.

class bindsnet.datasets.preprocess.**Rescale**(*output_size*)
Bases: object

Rescale image and bounding box. Args:

output_size (tuple or int): Desired output size. If int, square crop is made.

bindsnet.datasets.preprocess.**bgr2rgb**(*image*)

bindsnet.datasets.preprocess.**binary_image**(*image: numpy.ndarray*) → numpy.ndarray
Converts input image into black and white (binary)

Parameters **image** – Gray-scaled image.

Returns Black and white image.

bindsnet.datasets.preprocess.**computeCropPadImageLocation**(*bbox_tight*, *image*)

bindsnet.datasets.preprocess.**crop**(*image: numpy.ndarray*, *x1: int*, *x2: int*, *y1: int*, *y2: int*) → numpy.ndarray
Crops an image given coordinates of cropping box.

Parameters

- **image** – 3-dimensional image.
- **x1** – Left x coordinate.
- **x2** – Right x coordinate.
- **y1** – Bottom y coordinate.
- **y2** – Top y coordinate.

Returns Image cropped using coordinates (x1, x2, y1, y2).

bindsnet.datasets.preprocess.**cropPadImage**(*bbox_tight*, *image*)

bindsnet.datasets.preprocess.**crop_sample**(*sample*)
Given a sample image with bounding box, this method returns the image crop at the bounding box location with twice the width and height for context.

bindsnet.datasets.preprocess.**gray_scale**(*image: numpy.ndarray*) → numpy.ndarray
Converts RGB image into grayscale.

Parameters **image** – RGB image.

Returns Gray-scaled image.

bindsnet.datasets.preprocess.**sample_exp_two_sides**(*lambda_*)

bindsnet.datasets.preprocess.**sample_rand_uniform**()

bindsnet.datasets.preprocess.**shift_crop_training_sample**(*sample*, *bb_params*)
Given an image with bounding box, this method randomly shifts the box and generates a training example. It returns current image crop with shifted box (with respect to current image).

bindsnet.datasets.preprocess.**subsample**(*image: numpy.ndarray*, *x: int*, *y: int*) → numpy.ndarray
Scale the image to (x, y).

Parameters

- **image** – Image to be rescaled.
- **x** – Output value for `image`'s x dimension.
- **y** – Output value for `image`'s y dimension.

Returns Re-scaled image.

bindsnet.datasets.spoken_mnist module

class bindsnet.datasets.spoken_mnist.**SpokenMNIST**(*path: str*, *download: bool = False*, *shuffle: bool = True*, *train: bool = True*, *split: float = 0.8*, *num_samples: int = -1*)

Bases: `torch.utils.data.dataset.Dataset`

Handles loading and saving of the Spoken MNIST audio dataset (link).

Constructor for the `SpokenMNIST` object. Makes the data directory if it doesn't already exist.

Parameters

- **path** – Pathname of directory in which to store the dataset.
- **download** – Whether or not to download the dataset (requires internet connection).
- **shuffle** – Whether to randomly permute order of dataset.
- **train** – Load training split if true else load test split
- **split** – Train, test split; in range `(0, 1)`.
- **num_samples** – Number of samples to pass to the batch

digit = 9

example = 49

files = ['0_jackson_0.wav', '0_jackson_1.wav', '0_jackson_2.wav', '0_jackson_3.wav', '

n_files = 1500

process_data(*file_names: Iterable[str]*) → Tuple[List[torch.Tensor], torch.Tensor]

Opens files of Spoken MNIST data and processes them into `numpy` arrays.

Parameters **file_names** – Names of the files containing Spoken MNIST audio to load.

Returns Processed Spoken MNIST audio and label data.

speaker = 'theo'

test_pickle = 'test.pt'

train_pickle = 'train.pt'

url = 'https://github.com/Jakobovski/free-spoken-digit-dataset/archive/master.zip'

bindsnet.datasets.torchvision_wrapper module

bindsnet.datasets.torchvision_wrapper.**create_torchvision_dataset_wrapper**(*ds_type*)

Creates wrapper classes for datasets that output `(image, label)` from `__getitem__`. This applies to all of the datasets inside of `torchvision`.

Module contents

bindsnet.datasets.**create_torchvision_dataset_wrapper**(*ds_type*)
Creates wrapper classes for datasets that output `(image, label)` from `__getitem__`. This applies to all of the datasets inside of `torchvision`.

class bindsnet.datasets.**SpokenMNIST**(*path: str*, *download: bool = False*, *shuffle: bool = True*, *train: bool = True*, *split: float = 0.8*, *num_samples: int = -1*)
Bases: `torch.utils.data.dataset.Dataset`

Handles loading and saving of the Spoken MNIST audio dataset (link).

Constructor for the `SpokenMNIST` object. Makes the data directory if it doesn't already exist.

Parameters

- **path** – Pathname of directory in which to store the dataset.
- **download** – Whether or not to download the dataset (requires internet connection).
- **shuffle** – Whether to randomly permute order of dataset.
- **train** – Load training split if true else load test split
- **split** – Train, test split; in range `(0, 1)`.
- **num_samples** – Number of samples to pass to the batch

digit = 9

example = 49

files = ['0_jackson_0.wav', '0_jackson_1.wav', '0_jackson_2.wav', '0_jackson_3.wav', '

n_files = 1500

process_data(*file_names: Iterable[str]*) → Tuple[List[torch.Tensor], torch.Tensor]
Opens files of Spoken MNIST data and processes them into `numpy` arrays.

Parameters **file_names** – Names of the files containing Spoken MNIST audio to load.

Returns Processed Spoken MNIST audio and label data.

speaker = 'theo'

test_pickle = 'test.pt'

train_pickle = 'train.pt'

url = 'https://github.com/Jakobovski/free-spoken-digit-dataset/archive/master.zip'

class bindsnet.datasets.**Davis**(*root*, *task='unsupervised'*, *subset='train'*, *sequences='all'*, *resolution='480p'*, *size=(600, 480)*, *codalab=False*, *download=False*, *num_samples: int = -1*)
Bases: `torch.utils.data.dataset.Dataset`

Class to read the DAVIS dataset :param root: Path to the DAVIS folder that contains JPEGImages, Annotations,

etc. folders.

Parameters

- **task** – Task to load the annotations, choose between semi-supervised or unsupervised.
- **subset** – Set to load the annotations
- **sequences** – Sequences to consider, 'all' to use all the sequences in a set.

- **resolution** – Specify the resolution to use the dataset, choose between '480' and 'Full-Resolution'
- **download** – Specify whether to download the dataset if it is not present
- **num_samples** – Number of samples to pass to the batch

DATASET_WEB = 'https://davischallenge.org/davis2017/code.html'

RESOLUTION_OPTIONS = ['480p', 'Full-Resolution']

SUBSET_OPTIONS = ['train', 'val', 'test-dev', 'test-challenge']

TASKS = ['semi-supervised', 'unsupervised']

VOID_LABEL = 255

get_all_images(*sequence*)

get_all_masks(*sequence*, *separate_objects_masks=False*)

get_frames(*sequence*)

get_sequences()

static progress(*count*, *block_size*, *total_size*)
Simple progress indicator for the download of the dataset.

class bindsnet.datasets.**ALOV300**(*root*, *transform*, *input_size*, *download=False*)
Bases: torch.utils.data.dataset.Dataset

Class to read the ALOV dataset

Parameters

- **root** – Path to the ALOV folder that contains JPEGImages, annotations, etc. folders.
- **input_size** – The input size of network that is using this data, for rescaling.
- **download** – Specify whether to download the dataset if it is not present.
- **num_samples** – Number of samples to pass to the batch

DATASET_WEB = 'http://alov300pp.joomlafree.it/dataset-resources.html'

VOID_LABEL = 255

get_bb(*ann*)
Parses ALOV annotation and returns bounding box in the format: [left, upper, width, height]

get_orig_sample(*idx*, *i=1*)
Returns original image with bounding box at a specific index. Range of valid index: [0, self.len-1].

get_sample(*idx*)
Returns sample without transformation for visualization.

Sample consists of resized previous and current frame with target which is passed to the network. Bounding box values are normalized between 0 and 1 with respect to the target frame and then scaled by factor of 10.

progress(*count*, *block_size*, *total_size*)

show(*idx*, *is_current=1*)
Helper function to display image at a particular index with grounttruth bounding box.

Arguments: idx: index is_current: 0 for previous frame and 1 for current frame

show_sample(*idx*)
Helper function to display sample, which is passed to GOTURN. Shows previous frame and current frame with bounding box.

bindsnet.datasets.**time_aware_collate**(*batch*)
Puts each data field into a tensor with dimensions `[time, batch size, ...]`

Interpretation of dimensions being input: - 0 dim (,) - (1, batch_size, 1) - 1 dim (time,) - (time, batch_size, 1) - >2 dim (time, n_0, ...) - (time, batch_size, n_0, ...)

class bindsnet.datasets.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=<function time_aware_collate>*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*)
Bases: `torch.utils.data.dataloader.DataLoader`

bindsnet.datasets.**CIFAR10**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**CIFAR100**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**Cityscapes**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**CocoCaptions**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**CocoDetection**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**DatasetFolder**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**EMNIST**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**FakeData**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**FashionMNIST**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**Flickr30k**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**Flickr8k**
alias of `bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper`

bindsnet.datasets.**ImageFolder**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**KMNIST**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**LSUN**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**LSUNClass**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**MNIST**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**Omniglot**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**PhotoTour**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**SBU**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**SEMEION**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**STL10**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**SVHN**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**VOCDetection**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

bindsnet.datasets.**VOCSegmentation**
alias of bindsnet.datasets.torchvision_wrapper.create_torchvision_dataset_wrapper.<locals>.TorchvisionDatasetWrapper

4.1.4 bindsnet.encoding package

Submodules

bindsnet.encoding.encoders module

class bindsnet.encoding.encoders.**BernoulliEncoder**(*time: int*, *dt: float = 1.0*, ***kwargs*)
Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable BernoulliEncoder which encodes as defined in bindsnet.encoding.bernoulli

Parameters

- **time** – Length of Bernoulli spike train per input variable.
- **dt** – Simulation time step.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per time step.

class bindsnet.encoding.encoders.**Encoder**(**args*, ***kwargs*)
Bases: object

Base class for spike encodings transforms.

Calls self.enc from the subclass and passes whatever arguments were provided. self.enc must be callable with torch.Tensor, *args, **kwargs

class bindsnet.encoding.encoders.**NullEncoder**
Bases: *bindsnet.encoding.encoders.Encoder*

Pass through of the datum that was input.

Note: This is not a real spike encoder. Be careful with the usage of this class.

---

class bindsnet.encoding.encoders.**PoissonEncoder**(*time: int*, *dt: float = 1.0*, *approx: bool = False*, ***kwargs*)
Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable PoissonEncoder which encodes as defined in ‘‘bindsnet.encoding.poisson‘

Parameters

- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.
- **approx** – Bool: use alternate faster, less accurate computation.

class bindsnet.encoding.encoders.**RankOrderEncoder**(*time: int*, *dt: float = 1.0*, ***kwargs*)
Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable RankOrderEncoder which encodes as defined in bindsnet.encoding.rank_order

Parameters

- **time** – Length of RankOrder spike train per input variable.
- **dt** – Simulation time step.

class bindsnet.encoding.encoders.**RepeatEncoder**(*time: int*, *dt: float = 1.0*, ***kwargs*)
Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable RepeatEncoder which encodes as defined in bindsnet.encoding.repeat

Parameters

- **time** – Length of repeat spike train per input variable.

- **dt** – Simulation time step.

class bindsnet.encoding.encoders.**SingleEncoder**(*time: int*, *dt: float = 1.0*, *sparsity: float = 0.5*, ***kwargs*)

Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable SingleEncoder which encodes as defined in `bindsnet.encoding.single`

Parameters

- **time** – Length of single spike train per input variable.
- **dt** – Simulation time step.
- **sparsity** – Sparsity of the input representation. 0 for no spikes and 1 for all spikes.

bindsnet.encoding.encodings module

bindsnet.encoding.encodings.**bernoulli**(*datum: torch.Tensor*, *time: Optional[int] = None*, *dt: float = 1.0*, *device='cpu'*, ***kwargs*) → torch.Tensor

Generates Bernoulli-distributed spike trains based on input intensity. Inputs must be non-negative. Spikes correspond to successful Bernoulli trials, with success probability equal to (normalized in [0, 1]) input value.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of Bernoulli spike train per input variable.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of Bernoulli-distributed spikes.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per Bernoulli trial.

bindsnet.encoding.encodings.**poisson**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, *device='cpu'*, *approx=False*, ***kwargs*) → torch.Tensor

Generates Poisson-distributed spike trains based on input intensity. Inputs must be non-negative, and give the firing rate in Hz. Inter-spike intervals (ISIs) for non-negative data incremented by one to avoid zero intervals while maintaining ISI distributions.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.
- **device** – target destination of poisson spikes.
- **approx** – Bool: use alternate faster, less accurate computation.

Returns Tensor of shape `[time, n_1, ..., n_k]` of Poisson-distributed spikes.

bindsnet.encoding.encodings.**rank_order**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, ***kwargs*) → torch.Tensor

Encodes data via a rank order coding-like representation. One spike per neuron, temporally ordered by decreasing intensity. Inputs must be non-negative.

Parameters

- **datum** – Tensor of shape `[n_samples, n_1, ..., n_k]`.

- **time** – Length of rank order-encoded spike train per input variable.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of rank order-encoded spikes.

`bindsnet.encoding.encodings.`**`repeat`**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, ***kwargs*) → torch.Tensor

Parameters

- **datum** – Repeats a tensor along a new dimension in the 0th position for `int(time / dt)` timesteps.
- **time** – Tensor of shape `[n_1, ..., n_k]`.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of repeated data along the 0-th dimension.

`bindsnet.encoding.encodings.`**`single`**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, *sparsity: float = 0.5*, *device='cpu'*, ***kwargs*) → torch.Tensor

Generates timing based single-spike encoding. Spike occurs earlier if the intensity of the input feature is higher. Features whose value is lower than the threshold remain silent.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of the input and output.
- **dt** – Simulation time step.
- **sparsity** – Sparsity of the input representation. 0 for no spikes and 1 for all spikes.

Returns Tensor of shape `[time, n_1, ..., n_k]`.

bindsnet.encoding.loaders module

`bindsnet.encoding.loaders.`**`bernoulli_loader`**(*data: Union[torch.Tensor, Iterable[torch.Tensor]]*, *time: Optional[int] = None*, *dt: float = 1.0*, ***kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.bernoulli` to iteratively encode a sequence of data.

Parameters

- **data** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **time** – Length of Bernoulli spike train per input variable.
- **dt** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of Bernoulli-distributed spikes.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per Bernoulli trial.

`bindsnet.encoding.loaders.`**`poisson_loader`**(*data: Union[torch.Tensor, Iterable[torch.Tensor]]*, *time: int*, *dt: float = 1.0*, ***kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.poisson` to iteratively encode a sequence of data.

Parameters

- **data** – Tensor of shape `[n_samples, n_1, ..., n_k]`.

- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of Poisson-distributed spikes.

`bindsnet.encoding.loaders.`**rank_order_loader**(*data: Union[torch.Tensor, Iterable[torch.Tensor]], time: int, dt: float = 1.0, **kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.rank_order` to iteratively encode a sequence of data.

Parameters

- **data** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **time** – Length of rank order-encoded spike train per input variable.
- **dt** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of rank order-encoded spikes.

Module contents

`bindsnet.encoding.`**single**(*datum: torch.Tensor, time: int, dt: float = 1.0, sparsity: float = 0.5, device='cpu', **kwargs*) → torch.Tensor

Generates timing based single-spike encoding. Spike occurs earlier if the intensity of the input feature is higher. Features whose value is lower than the threshold remain silent.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of the input and output.
- **dt** – Simulation time step.
- **sparsity** – Sparsity of the input representation. 0 for no spikes and 1 for all spikes.

Returns Tensor of shape `[time, n_1, ..., n_k]`.

`bindsnet.encoding.`**repeat**(*datum: torch.Tensor, time: int, dt: float = 1.0, **kwargs*) → torch.Tensor

Parameters

- **datum** – Repeats a tensor along a new dimension in the 0th position for `int(time / dt)` timesteps.
- **time** – Tensor of shape `[n_1, ..., n_k]`.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of repeated data along the 0-th dimension.

`bindsnet.encoding.`**bernoulli**(*datum: torch.Tensor, time: Optional[int] = None, dt: float = 1.0, device='cpu', **kwargs*) → torch.Tensor

Generates Bernoulli-distributed spike trains based on input intensity. Inputs must be non-negative. Spikes correspond to successful Bernoulli trials, with success probability equal to (normalized in [0, 1]) input value.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of Bernoulli spike train per input variable.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of Bernoulli-distributed spikes.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per Bernoulli trial.

bindsnet.encoding.**poisson**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, *device='cpu'*, *approx=False*, ***kwargs*) → torch.Tensor

Generates Poisson-distributed spike trains based on input intensity. Inputs must be non-negative, and give the firing rate in Hz. Inter-spike intervals (ISIs) for non-negative data incremented by one to avoid zero intervals while maintaining ISI distributions.

Parameters

- **datum** – Tensor of shape `[n_1, ..., n_k]`.
- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.
- **device** – target destination of poisson spikes.
- **approx** – Bool: use alternate faster, less accurate computation.

Returns Tensor of shape `[time, n_1, ..., n_k]` of Poisson-distributed spikes.

bindsnet.encoding.**rank_order**(*datum: torch.Tensor*, *time: int*, *dt: float = 1.0*, ***kwargs*) → torch.Tensor

Encodes data via a rank order coding-like representation. One spike per neuron, temporally ordered by decreasing intensity. Inputs must be non-negative.

Parameters

- **datum** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **time** – Length of rank order-encoded spike train per input variable.
- **dt** – Simulation time step.

Returns Tensor of shape `[time, n_1, ..., n_k]` of rank order-encoded spikes.

bindsnet.encoding.**bernoulli_loader**(*data: Union[torch.Tensor, Iterable[torch.Tensor]]*, *time: Optional[int] = None*, *dt: float = 1.0*, ***kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.bernoulli` to iteratively encode a sequence of data.

Parameters

- **data** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **time** – Length of Bernoulli spike train per input variable.
- **dt** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of Bernoulli-distributed spikes.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per Bernoulli trial.

bindsnet.encoding.**poisson_loader**(*data: Union[torch.Tensor, Iterable[torch.Tensor]]*, *time: int*, *dt: float = 1.0*, ***kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.poisson` to iteratively encode a sequence of data.

Parameters

- **data** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of Poisson-distributed spikes.

`bindsnet.encoding.`**`rank_order_loader`**(*data: Union[torch.Tensor, Iterable[torch.Tensor]], time: int, dt: float = 1.0, **kwargs*) → Iterator[torch.Tensor]

Lazily invokes `bindsnet.encoding.rank_order` to iteratively encode a sequence of data.

Parameters

- **`data`** – Tensor of shape `[n_samples, n_1, ..., n_k]`.
- **`time`** – Length of rank order-encoded spike train per input variable.
- **`dt`** – Simulation time step.

Returns Tensors of shape `[time, n_1, ..., n_k]` of rank order-encoded spikes.

class `bindsnet.encoding.`**`Encoder`**(**args*, ***kwargs*)

Bases: `object`

Base class for spike encodings transforms.

Calls `self.enc` from the subclass and passes whatever arguments were provided. `self.enc` must be callable with `torch.Tensor`, `*args`, `**kwargs`

class `bindsnet.encoding.`**`NullEncoder`**

Bases: *`bindsnet.encoding.encoders.Encoder`*

Pass through of the datum that was input.

Note: This is not a real spike encoder. Be careful with the usage of this class.

---

class `bindsnet.encoding.`**`SingleEncoder`**(*time: int*, *dt: float = 1.0*, *sparsity: float = 0.5*, ***kwargs*)

Bases: *`bindsnet.encoding.encoders.Encoder`*

Creates a callable SingleEncoder which encodes as defined in `bindsnet.encoding.single`

Parameters

- **`time`** – Length of single spike train per input variable.
- **`dt`** – Simulation time step.
- **`sparsity`** – Sparsity of the input representation. 0 for no spikes and 1 for all spikes.

class `bindsnet.encoding.`**`RepeatEncoder`**(*time: int*, *dt: float = 1.0*, ***kwargs*)

Bases: *`bindsnet.encoding.encoders.Encoder`*

Creates a callable `RepeatEncoder` which encodes as defined in `bindsnet.encoding.repeat`

Parameters

- **`time`** – Length of repeat spike train per input variable.
- **`dt`** – Simulation time step.

class `bindsnet.encoding.`**`BernoulliEncoder`**(*time: int*, *dt: float = 1.0*, ***kwargs*)

Bases: *`bindsnet.encoding.encoders.Encoder`*

Creates a callable `BernoulliEncoder` which encodes as defined in `bindsnet.encoding.bernoulli`

Parameters

- **`time`** – Length of Bernoulli spike train per input variable.
- **`dt`** – Simulation time step.

Keyword arguments:

Parameters **max_prob** (*float*) – Maximum probability of spike per time step.

class bindsnet.encoding.**PoissonEncoder**(*time: int*, *dt: float = 1.0*, *approx: bool = False*, ***kwargs*)

Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable PoissonEncoder which encodes as defined in ‘‘bindsnet.encoding.poisson‘

Parameters

- **time** – Length of Poisson spike train per input variable.
- **dt** – Simulation time step.
- **approx** – Bool: use alternate faster, less accurate computation.

class bindsnet.encoding.**RankOrderEncoder**(*time: int*, *dt: float = 1.0*, ***kwargs*)

Bases: *bindsnet.encoding.encoders.Encoder*

Creates a callable RankOrderEncoder which encodes as defined in `bindsnet.encoding.rank_order`

Parameters

- **time** – Length of RankOrder spike train per input variable.
- **dt** – Simulation time step.

4.1.5 bindsnet.environment package

Submodules

bindsnet.environment.environment module

class bindsnet.environment.environment.**Environment**

Bases: `abc.ABC`

Abstract environment class.

close() → None
Abstract method header for `close()`.

preprocess() → None
Abstract method header for `preprocess()`.

render() → None
Abstract method header for `render()`.

reset() → None
Abstract method header for `reset()`.

step(*a: int*) → Tuple[Any, ...]
Abstract method head for `step()`.

Parameters **a** – Integer action to take in environment.

class bindsnet.environment.environment.**GymEnvironment**(*name: str*, *encoder: bindsnet.encoding.encoders.Encoder = <bindsnet.encoding.encoders.NullEncoder object>*, ***kwargs*)

Bases: *bindsnet.environment.environment.Environment*

A wrapper around the OpenAI `gym` environments.

Initializes the environment wrapper. This class makes the assumption that the OpenAI `gym` environment will provide an image of format HxW or CxHxW as an observation (we will add the C dimension to HxW tensors) or a 1D observation in which case no dimensions will be added.

Parameters

- **name** – The name of an OpenAI `gym` environment.
- **encoder** – Function to encode observations into spike trains.

Keyword arguments:

Parameters

- **max_prob** (*float*) – Maximum spiking probability.
- **clip_rewards** (*bool*) – Whether or not to use `np.sign` of rewards.
- **history** (*int*) – Number of observations to keep track of.
- **delta** (*int*) – Step size to save observations in history.
- **add_channel_dim** (*bool*) – Allows for the adding of the channel dimension in 2D inputs.

close() → None
Wrapper around the OpenAI `gym` environment `close()` function.

preprocess() → None
Pre-processing step for an observation from a `gym` environment.

render() → None
Wrapper around the OpenAI `gym` environment `render()` function.

reset() → torch.Tensor
Wrapper around the OpenAI `gym` environment `reset()` function.

Returns Observation from the environment.

step(*a: int*) → Tuple[torch.Tensor, float, bool, Dict[Any, Any]]
Wrapper around the OpenAI `gym` environment `step()` function.

Parameters **a** – Action to take in the environment.

Returns Observation, reward, done flag, and information dictionary.

update_history() → None
Updates the observations inside history by performing subtraction from most recent observation and the sum of previous observations. If there are not enough observations to take a difference from, simply store the observation without any differencing.

update_index() → None
Updates the index to keep track of history. For example: `history = 4`, `delta = 3` will produce `self.history = {1, 4, 7, 10}` and `self.history_index` will be updated according to `self.delta` and will wrap around the history dictionary.

Module contents

class bindsnet.environment.**Environment**
Bases: `abc.ABC`

Abstract environment class.

close() → None
Abstract method header for close().

preprocess() → None
Abstract method header for preprocess().

render() → None
Abstract method header for render().

reset() → None
Abstract method header for reset().

step(*a: int*) → Tuple[Any, ...]
Abstract method head for step().

Parameters **a** – Integer action to take in environment.

class bindsnet.environment.**GymEnvironment**(*name: str*, *encoder: bindsnet.encoding.encoders.Encoder = <bindsnet.encoding.encoders.NullEncoder object>*, ***kwargs*)
Bases: *bindsnet.environment.environment.Environment*

A wrapper around the OpenAI gym environments.

Initializes the environment wrapper. This class makes the assumption that the OpenAI gym environment will provide an image of format HxW or CxHxW as an observation (we will add the C dimension to HxW tensors) or a 1D observation in which case no dimensions will be added.

Parameters

- **name** – The name of an OpenAI gym environment.
- **encoder** – Function to encode observations into spike trains.

Keyword arguments:

Parameters

- **max_prob** (*float*) – Maximum spiking probability.
- **clip_rewards** (*bool*) – Whether or not to use np.sign of rewards.
- **history** (*int*) – Number of observations to keep track of.
- **delta** (*int*) – Step size to save observations in history.
- **add_channel_dim** (*bool*) – Allows for the adding of the channel dimension in 2D inputs.

close() → None
Wrapper around the OpenAI gym environment close() function.

preprocess() → None
Pre-processing step for an observation from a gym environment.

render() → None
Wrapper around the OpenAI gym environment render() function.

reset() → torch.Tensor
Wrapper around the OpenAI gym environment reset() function.

Returns Observation from the environment.

step(*a: int*) → Tuple[torch.Tensor, float, bool, Dict[Any, Any]]
Wrapper around the OpenAI gym environment step() function.

Parameters **a** – Action to take in the environment.

Returns Observation, reward, done flag, and information dictionary.

update_history() → None
Updates the observations inside history by performing subtraction from most recent observation and the sum of previous observations. If there are not enough observations to take a difference from, simply store the observation without any differencing.

update_index() → None
Updates the index to keep track of history. For example: `history = 4`, `delta = 3` will produce `self.history = {1, 4, 7, 10}` and `self.history_index` will be updated according to `self.delta` and will wrap around the history dictionary.

4.1.6 bindsnet.evaluation package

Submodules

bindsnet.evaluation.evaluation module

bindsnet.evaluation.evaluation.**all_activity**(*spikes: torch.Tensor*, *assignments: torch.Tensor*, *n_labels: int*) → torch.Tensor
Classify data with the label with highest average spiking activity over all neurons.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a layer's spiking activity.
- **assignments** – A vector of shape `(n_neurons,)` of neuron label assignments.
- **n_labels** – The number of target labels in the data.

Returns Predictions tensor of shape `(n_samples,)` resulting from the "all activity" classification scheme.

bindsnet.evaluation.evaluation.**assign_labels**(*spikes: torch.Tensor*, *labels: torch.Tensor*, *n_labels: int*, *rates: Optional[torch.Tensor] = None*, *alpha: float = 1.0*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]
Assign labels to the neurons based on highest average spiking activity.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a single layer's spiking activity.
- **labels** – Vector of shape `(n_samples,)` with data labels corresponding to spiking activity.
- **n_labels** – The number of target labels in the data.
- **rates** – If passed, these represent spike rates from a previous `assign_labels()` call.
- **alpha** – Rate of decay of label assignments.

Returns Tuple of class assignments, per-class spike proportions, and per-class firing rates.

bindsnet.evaluation.evaluation.**logreg_fit**(*spikes: torch.Tensor*, *labels: torch.Tensor*, *logreg: sklearn.linear_model._logistic.LogisticRegression*) → sklearn.linear_model._logistic.LogisticRegression

(Re)fit logistic regression model to spike data summed over time.

Parameters

- **spikes** – Summed (over time) spikes of shape `(n_examples, time, n_neurons)`.
- **labels** – Vector of shape `(n_samples,)` with data labels corresponding to spiking activity.
- **logreg** – Logistic regression model from previous fits.

Returns (Re)fitted logistic regression model.

bindsnet.evaluation.evaluation.**logreg_predict**(*spikes: torch.Tensor*, *logreg: sklearn.linear_model._logistic.LogisticRegression*) → torch.Tensor

Predicts classes according to spike data summed over time.

Parameters

- **spikes** – Summed (over time) spikes of shape `(n_examples, time, n_neurons)`.
- **logreg** – Logistic regression model from previous fits.

Returns Predictions per example.

bindsnet.evaluation.evaluation.**ngram**(*spikes: torch.Tensor, ngram_scores: Dict[Tuple[int, ...], torch.Tensor], n_labels: int, n: int*) → torch.Tensor

Predicts between `n_labels` using `ngram_scores`.

Parameters

- **spikes** – Spikes of shape `(n_examples, time, n_neurons)`.
- **ngram_scores** – Previously recorded scores to update.
- **n_labels** – The number of target labels in the data.
- **n** – The max size of n-gram to use.

Returns Predictions per example.

bindsnet.evaluation.evaluation.**proportion_weighting**(*spikes: torch.Tensor*, *assignments: torch.Tensor*, *proportions: torch.Tensor*, *n_labels: int*) → torch.Tensor

Classify data with the label with highest average spiking activity over all neurons, weighted by class-wise proportion.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a single layer's spiking activity.
- **assignments** – A vector of shape `(n_neurons,)` of neuron label assignments.
- **proportions** – A matrix of shape `(n_neurons, n_labels)` giving the per-class proportions of neuron spiking activity.
- **n_labels** – The number of target labels in the data.

Returns Predictions tensor of shape `(n_samples,)` resulting from the "proportion weighting" classification scheme.

bindsnet.evaluation.evaluation.**update_ngram_scores**(*spikes: torch.Tensor, labels: torch.Tensor, n_labels: int, n: int, ngram_scores: Dict[Tuple[int, ...], torch.Tensor]*) → Dict[Tuple[int, ...], torch.Tensor]

Updates ngram scores by adding the count of each spike sequence of length n from the past `n_examples`.

Parameters

- **spikes** – Spikes of shape `(n_examples, time, n_neurons)`.
- **labels** – The ground truth labels of shape `(n_examples)`.
- **n_labels** – The number of target labels in the data.
- **n** – The max size of n-gram to use.
- **ngram_scores** – Previously recorded scores to update.

Returns Dictionary mapping n-grams to vectors of per-class spike counts.

Module contents

bindsnet.evaluation.**assign_labels**(*spikes: torch.Tensor*, *labels: torch.Tensor*, *n_labels: int*, *rates: Optional[torch.Tensor] = None*, *alpha: float = 1.0*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Assign labels to the neurons based on highest average spiking activity.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a single layer's spiking activity.
- **labels** – Vector of shape `(n_samples,)` with data labels corresponding to spiking activity.
- **n_labels** – The number of target labels in the data.
- **rates** – If passed, these represent spike rates from a previous `assign_labels()` call.
- **alpha** – Rate of decay of label assignments.

Returns Tuple of class assignments, per-class spike proportions, and per-class firing rates.

bindsnet.evaluation.**logreg_fit**(*spikes: torch.Tensor*, *labels: torch.Tensor*, *logreg: sklearn.linear_model._logistic.LogisticRegression*) → sklearn.linear_model._logistic.LogisticRegression

(Re)fit logistic regression model to spike data summed over time.

Parameters

- **spikes** – Summed (over time) spikes of shape `(n_examples, time, n_neurons)`.
- **labels** – Vector of shape `(n_samples,)` with data labels corresponding to spiking activity.
- **logreg** – Logistic regression model from previous fits.

Returns (Re)fitted logistic regression model.

bindsnet.evaluation.**logreg_predict**(*spikes: torch.Tensor*, *logreg: sklearn.linear_model._logistic.LogisticRegression*) → torch.Tensor

Predicts classes according to spike data summed over time.

Parameters

- **spikes** – Summed (over time) spikes of shape `(n_examples, time, n_neurons)`.
- **logreg** – Logistic regression model from previous fits.

Returns Predictions per example.

bindsnet.evaluation.**all_activity**(*spikes: torch.Tensor*, *assignments: torch.Tensor*, *n_labels: int*) → torch.Tensor

Classify data with the label with highest average spiking activity over all neurons.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a layer's spiking activity.
- **assignments** – A vector of shape `(n_neurons,)` of neuron label assignments.
- **n_labels** – The number of target labels in the data.

Returns Predictions tensor of shape `(n_samples,)` resulting from the "all activity" classification scheme.

bindsnet.evaluation.**proportion_weighting**(*spikes: torch.Tensor*, *assignments: torch.Tensor*, *proportions: torch.Tensor*, *n_labels: int*) → torch.Tensor

Classify data with the label with highest average spiking activity over all neurons, weighted by class-wise proportion.

Parameters

- **spikes** – Binary tensor of shape `(n_samples, time, n_neurons)` of a single layer's spiking activity.
- **assignments** – A vector of shape `(n_neurons,)` of neuron label assignments.
- **proportions** – A matrix of shape `(n_neurons, n_labels)` giving the per-class proportions of neuron spiking activity.
- **n_labels** – The number of target labels in the data.

Returns Predictions tensor of shape `(n_samples,)` resulting from the "proportion weighting" classification scheme.

bindsnet.evaluation.**ngram**(*spikes: torch.Tensor, ngram_scores: Dict[Tuple[int, ...], torch.Tensor], n_labels: int, n: int*) → torch.Tensor

Predicts between `n_labels` using `ngram_scores`.

Parameters

- **spikes** – Spikes of shape `(n_examples, time, n_neurons)`.
- **ngram_scores** – Previously recorded scores to update.
- **n_labels** – The number of target labels in the data.
- **n** – The max size of n-gram to use.

Returns Predictions per example.

bindsnet.evaluation.**update_ngram_scores**(*spikes: torch.Tensor, labels: torch.Tensor, n_labels: int, n: int, ngram_scores: Dict[Tuple[int, ...], torch.Tensor]*) → Dict[Tuple[int, ...], torch.Tensor]

Updates ngram scores by adding the count of each spike sequence of length n from the past `n_examples`.

Parameters

- **spikes** – Spikes of shape `(n_examples, time, n_neurons)`.
- **labels** – The ground truth labels of shape `(n_examples)`.
- **n_labels** – The number of target labels in the data.
- **n** – The max size of n-gram to use.
- **ngram_scores** – Previously recorded scores to update.

Returns Dictionary mapping n-grams to vectors of per-class spike counts.

4.1.7 bindsnet.learning package

Submodules

bindsnet.learning.learning module

class bindsnet.learning.learning.**Hebbian**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Simple Hebbian learning rule. Pre- and post-synaptic updates are both positive.

Constructor for `Hebbian` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `Hebbian` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

class bindsnet.learning.learning.**LearningRule**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: `abc.ABC`

Abstract base class for learning rules.

Abstract constructor for the `LearningRule` object.

Parameters

- **connection** – An `AbstractConnection` object.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

update() → None
Abstract method for a learning rule update.

class bindsnet.learning.learning.**MSTDP**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated STDP. Adapted from (Florian 2007).

Constructor for `MSTDP` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `MSTDP` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments:

Parameters

- **tc_plus** – Time constant for pre-synaptic firing trace.
- **tc_minus** – Time constant for post-synaptic firing trace.

class bindsnet.learning.learning.**MSTDPET**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated STDP with eligibility trace. Adapted from (Florian 2007).

Constructor for `MSTDPET` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `MSTDPET` learning rule will modify.

- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments: :param float tc_plus: Time constant for pre-synaptic firing trace. :param float tc_minus: Time constant for post-synaptic firing trace. :param float tc_e_trace: Time constant for the eligibility trace.

class bindsnet.learning.learning.**NoOp**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Learning rule with no effect.

Abstract constructor for the `LearningRule` object.

Parameters

- **connection** – An `AbstractConnection` object.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

update(***kwargs*) → None

Abstract method for a learning rule update.

class bindsnet.learning.learning.**PostPre**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Simple STDP rule involving both pre- and post-synaptic spiking activity. By default, pre-synaptic update is negative and the post-synaptic update is positive.

Constructor for `PostPre` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `PostPre` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

class bindsnet.learning.learning.**Rmax**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated learning rule derived from reward maximization principles. Adapted from (Vasilaki et al., 2009).

Constructor for `R-max` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `R-max` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments:

Parameters

- **tc_c** (*float*) – Time constant for balancing naive Hebbian and policy gradient learning.
- **tc_e_trace** (*float*) – Time constant for the eligibility trace.

class bindsnet.learning.learning.**WeightDependentPostPre**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

STDP rule involving both pre- and post-synaptic spiking activity. The post-synaptic update is positive and the pre- synaptic update is negative, and both are dependent on the magnitude of the synaptic weights.

Constructor for `WeightDependentPostPre` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `WeightDependentPostPre` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

bindsnet.learning.reward module

class bindsnet.learning.reward.**AbstractReward**
Bases: abc.ABC

Abstract base class for reward computation.

compute(***kwargs*) → None
Computes/modifies reward.

update(***kwargs*) → None
Updates internal variables needed to modify reward. Usually called once per episode.

class bindsnet.learning.reward.**MovingAvgRPE**(***kwargs*)
Bases: *bindsnet.learning.reward.AbstractReward*

Computes reward prediction error (RPE) based on an exponential moving average (EMA) of past rewards.

Constructor for EMA reward prediction error.

compute(***kwargs*) → torch.Tensor
Computes the reward prediction error using EMA.

Keyword arguments:

Parameters **torch.Tensor] reward**(*Union[float,*) – Current reward.

Returns Reward prediction error.

update(***kwargs*) → None
Updates the EMAs. Called once per episode.

Keyword arguments:

Parameters

- **torch.Tensor] accumulated_reward** (*Union[float,*) – Reward accumulated over one episode.
- **steps** (*int*) – Steps in that episode.
- **ema_window** (*float*) – Width of the averaging window.

Module contents

class bindsnet.learning.**LearningRule**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: abc.ABC

Abstract base class for learning rules.

Abstract constructor for the LearningRule object.

Parameters

- **connection** – An AbstractConnection object.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

update() → None
Abstract method for a learning rule update.

class bindsnet.learning.**NoOp**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.learning.learning.LearningRule*

Learning rule with no effect.

Abstract constructor for the `LearningRule` object.

Parameters

- **connection** – An `AbstractConnection` object.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

update(***kwargs*) → None
Abstract method for a learning rule update.

class bindsnet.learning.**PostPre**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.learning.learning.LearningRule*

Simple STDP rule involving both pre- and post-synaptic spiking activity. By default, pre-synaptic update is negative and the post-synaptic update is positive.

Constructor for `PostPre` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `PostPre` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

class bindsnet.learning.**WeightDependentPostPre**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.learning.learning.LearningRule*

STDP rule involving both pre- and post-synaptic spiking activity. The post-synaptic update is positive and the pre- synaptic update is negative, and both are dependent on the magnitude of the synaptic weights.

Constructor for `WeightDependentPostPre` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `WeightDependentPostPre` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

class bindsnet.learning.**Hebbian**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Simple Hebbian learning rule. Pre- and post-synaptic updates are both positive.

Constructor for `Hebbian` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `Hebbian` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the batch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

class bindsnet.learning.**MSTDP**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated STDP. Adapted from (Florian 2007).

Constructor for `MSTDP` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `MSTDP` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments:

Parameters

- **tc_plus** – Time constant for pre-synaptic firing trace.
- **tc_minus** – Time constant for post-synaptic firing trace.

class bindsnet.learning.**MSTDPET**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated STDP with eligibility trace. Adapted from (Florian 2007).

Constructor for `MSTDPET` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `MSTDPET` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments: :param float tc_plus: Time constant for pre-synaptic firing trace. :param float tc_minus: Time constant for post-synaptic firing trace. :param float tc_e_trace: Time constant for the eligibility trace.

class bindsnet.learning.**Rmax**(*connection: bindsnet.network.topology.AbstractConnection, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.learning.learning.LearningRule*

Reward-modulated learning rule derived from reward maximization principles. Adapted from (Vasilaki et al., 2009).

Constructor for `R-max` learning rule.

Parameters

- **connection** – An `AbstractConnection` object whose weights the `R-max` learning rule will modify.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Coefficient controlling rate of decay of the weights each iteration.

Keyword arguments:

Parameters

- **tc_c** (*float*) – Time constant for balancing naive Hebbian and policy gradient learning.
- **tc_e_trace** (*float*) – Time constant for the eligibility trace.

4.1.8 bindsnet.models package

Submodules

bindsnet.models.models module

class bindsnet.models.models.**DiehlAndCook2015**(*n_inpt: int, n_neurons: int = 100, exc: float = 22.5, inh: float = 17.5, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: float = 0.0, wmax: float = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Implements the spiking neural network architecture from (Diehl & Cook 2015).

Constructor for class `DiehlAndCook2015`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **exc** – Strength of synapse weights from excitatory to inhibitory layer.
- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.models.**DiehlAndCook2015v2**(*n_inpt: int, n_neurons: int = 100, inh: float = 17.5, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: Optional[float] = 0.0, wmax: Optional[float] = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Slightly modifies the spiking neural network architecture from (Diehl & Cook 2015) by removing the inhibitory layer and replacing it with a recurrent inhibitory connection in the output layer (what used to be the excitatory layer).

Constructor for class `DiehlAndCook2015v2`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.models.**IncreasingInhibitionNetwork**(*n_input: int, n_neurons: int = 100, start_inhib: float = 1.0, max_inhib: float = 100.0, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: float = 0.0, wmax: float = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Implements the inhibitory layer structure of the spiking neural network architecture from (Hazan et al. 2018)

Constructor for class `IncreasingInhibitionNetwork`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.

- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.models.**LocallyConnectedNetwork**(*n_inpt: int, input_shape: List[int], kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]], n_filters: int, inh: float = 25.0, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, wmin: float = 0.0, wmax: float = 1.0, norm: Optional[float] = 0.2*)

Bases: *bindsnet.network.network.Network*

Defines a two-layer network in which the input layer is "locally connected" to the output layer, and the output layer is recurrently inhibited connected such that neurons with the same input receptive field inhibit each other.

Constructor for class `LocallyConnectedNetwork`. Uses `DiehlAndCookNodes` to avoid multiple spikes per timestep in the output layer population.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **input_shape** – Two-dimensional shape of input population.
- **kernel_size** – Size of input windows. Integer or two-tuple of integers.
- **stride** – Length of horizontal, vertical stride across input space. Integer or two-tuple of integers.
- **n_filters** – Number of locally connected filters per input region. Integer or two-tuple of integers.
- **inh** – Strength of synapse weights from output layer back onto itself.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on `Input` to `DiehlAndCookNodes` synapses.
- **wmax** – Maximum allowed weight on `Input` to `DiehlAndCookNodes` synapses.

- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **norm** – `Input` to `DiehlAndCookNodes` layer connection weights normalization constant.

class bindsnet.models.models.**TwoLayerNetwork**(*n_inpt: int, n_neurons: int = 100, dt: float = 1.0, wmin: float = 0.0, wmax: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, norm: float = 78.4*)

Bases: *bindsnet.network.network.Network*

Implements an `Input` instance connected to a `LIFNodes` instance with a fully-connected `Connection`.

Constructor for class `TwoLayerNetwork`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of neurons in the `LIFNodes` population.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on `Input` to `LIFNodes` synapses.
- **wmax** – Maximum allowed weight on `Input` to `LIFNodes` synapses.
- **norm** – `Input` to `LIFNodes` layer connection weights normalization constant.

Module contents

class bindsnet.models.**TwoLayerNetwork**(*n_inpt: int, n_neurons: int = 100, dt: float = 1.0, wmin: float = 0.0, wmax: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, norm: float = 78.4*)

Bases: *bindsnet.network.network.Network*

Implements an `Input` instance connected to a `LIFNodes` instance with a fully-connected `Connection`.

Constructor for class `TwoLayerNetwork`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of neurons in the `LIFNodes` population.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on `Input` to `LIFNodes` synapses.
- **wmax** – Maximum allowed weight on `Input` to `LIFNodes` synapses.

- **norm** – Input to LIFNodes layer connection weights normalization constant.

class bindsnet.models.**DiehlAndCook2015v2**(*n_inpt: int, n_neurons: int = 100, inh: float = 17.5, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: Optional[float] = 0.0, wmax: Optional[float] = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Slightly modifies the spiking neural network architecture from (Diehl & Cook 2015) by removing the inhibitory layer and replacing it with a recurrent inhibitory connection in the output layer (what used to be the excitatory layer).

Constructor for class DiehlAndCook2015v2.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of DiehlAndCookNodes membrane threshold potential.
- **tc_theta_decay** – Time constant of DiehlAndCookNodes threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.**DiehlAndCook2015**(*n_inpt: int, n_neurons: int = 100, exc: float = 22.5, inh: float = 17.5, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: float = 0.0, wmax: float = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Implements the spiking neural network architecture from (Diehl & Cook 2015).

Constructor for class DiehlAndCook2015.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **exc** – Strength of synapse weights from excitatory to inhibitory layer.

- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.**IncreasingInhibitionNetwork**(*n_input: int, n_neurons: int = 100, start_inhib: float = 1.0, max_inhib: float = 100.0, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, wmin: float = 0.0, wmax: float = 1.0, norm: float = 78.4, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, inpt_shape: Optional[Iterable[int]] = None*)

Bases: *bindsnet.network.network.Network*

Implements the inhibitory layer structure of the spiking neural network architecture from (Hazan et al. 2018)

Constructor for class `IncreasingInhibitionNetwork`.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **n_neurons** – Number of excitatory, inhibitory neurons.
- **inh** – Strength of synapse weights from inhibitory to excitatory layer.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on input to excitatory synapses.
- **wmax** – Maximum allowed weight on input to excitatory synapses.
- **norm** – Input to excitatory layer connection weights normalization constant.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **inpt_shape** – The dimensionality of the input layer.

class bindsnet.models.**LocallyConnectedNetwork**(*n_inpt: int, input_shape: List[int], kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]], n_filters: int, inh: float = 25.0, dt: float = 1.0, nu: Union[float, Sequence[float], None] = (0.0001, 0.01), reduction: Optional[callable] = None, theta_plus: float = 0.05, tc_theta_decay: float = 10000000.0, wmin: float = 0.0, wmax: float = 1.0, norm: Optional[float] = 0.2*)

Bases: *bindsnet.network.network.Network*

Defines a two-layer network in which the input layer is "locally connected" to the output layer, and the output layer is recurrently inhibited connected such that neurons with the same input receptive field inhibit each other.

Constructor for class `LocallyConnectedNetwork`. Uses `DiehlAndCookNodes` to avoid multiple spikes per timestep in the output layer population.

Parameters

- **n_inpt** – Number of input neurons. Matches the 1D size of the input data.
- **input_shape** – Two-dimensional shape of input population.
- **kernel_size** – Size of input windows. Integer or two-tuple of integers.
- **stride** – Length of horizontal, vertical stride across input space. Integer or two-tuple of integers.
- **n_filters** – Number of locally connected filters per input region. Integer or two-tuple of integers.
- **inh** – Strength of synapse weights from output layer back onto itself.
- **dt** – Simulation time step.
- **nu** – Single or pair of learning rates for pre- and post-synaptic events, respectively.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **wmin** – Minimum allowed weight on `Input` to `DiehlAndCookNodes` synapses.
- **wmax** – Maximum allowed weight on `Input` to `DiehlAndCookNodes` synapses.
- **theta_plus** – On-spike increment of `DiehlAndCookNodes` membrane threshold potential.
- **tc_theta_decay** – Time constant of `DiehlAndCookNodes` threshold potential decay.
- **norm** – `Input` to `DiehlAndCookNodes` layer connection weights normalization constant.

4.1.9 bindsnet.network package

Submodules

bindsnet.network.monitors module

class bindsnet.network.monitors.**AbstractMonitor**

Bases: `abc.ABC`

Abstract base class for state variable monitors.

class bindsnet.network.monitors.**Monitor**(*obj: Union[bindsnet.network.nodes.Nodes, bindsnet.network.topology.AbstractConnection], state_vars: Iterable[str], time: Optional[int] = None, batch_size: int = 1, device: str = 'cpu'*)

Bases: *bindsnet.network.monitors.AbstractMonitor*

Records state variables of interest.

Constructs a `Monitor` object.

Parameters

- **obj** – An object to record state variables from during network simulation.
- **state_vars** – Iterable of strings indicating names of state variables to record.
- **time** – If not `None`, pre-allocate memory for state variable recording.
- **device** – Allow the monitor to be on different device separate from Network device

get(*var: str*) → torch.Tensor

Return recording to user.

Parameters **var** – State variable recording to return.

Returns Tensor of shape `[time, n_1, ..., n_k]`, where `[n_1, ..., n_k]` is the shape of the recorded state

variable. Note, if time == *None*, get return the logs and empty the monitor variable

record() → None

Appends the current value of the recorded state variables to the recording.

reset_state_variables() → None

Resets recordings to empty ‘‘List‘‘s.

class bindsnet.network.monitors.**NetworkMonitor**(*network: Network, layers: Optional[Iterable[str]] = None, connections: Optional[Iterable[str]] = None, state_vars: Optional[Iterable[str]] = None, time: Optional[int] = None*)

Bases: *bindsnet.network.monitors.AbstractMonitor*

Record state variables of all layers and connections.

Constructs a `NetworkMonitor` object.

Parameters

- **network** – Network to record state variables from.
- **layers** – Layers to record state variables from.
- **connections** – Connections to record state variables from.
- **state_vars** – List of strings indicating names of state variables to record.
- **time** – If not `None`, pre-allocate memory for state variable recording.

get() → Dict[str, Dict[str, Union[bindsnet.network.nodes.Nodes, bindsnet.network.topology.AbstractConnection]]]

Return entire recording to user.

Returns Dictionary of dictionary of all layers’ and connections’ recorded state variables.

record() → None
Appends the current value of the recorded state variables to the recording.

reset_state_variables() → None
Resets recordings to empty `torch.Tensors`.

save(*path: str*, *fmt: str = 'npz'*) → None
Write the recording dictionary out to file.

Parameters

- **path** – The directory to which to write the monitor's recording.
- **fmt** – Type of file to write to disk. One of `"pickle"` or `"npz"`.

bindsnet.network.network module

class bindsnet.network.network.**Network**(*dt: float = 1.0*, *batch_size: int = 1*, *learning: bool = True*, *reward_fn: Optional[Type[bindsnet.learning.reward.AbstractReward]] = None*)

Bases: `torch.nn.modules.module.Module`

Central object of the `bindsnet` package. Responsible for the simulation and interaction of nodes and connections.

Example:

```
import torch
import matplotlib.pyplot as plt

from bindsnet          import encoding
from bindsnet.network import Network, nodes, topology, monitors

network = Network(dt=1.0)  # Instantiates network.

X = nodes.Input(100)  # Input layer.
Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  # Connection␣
↪from X to Y.

# Spike monitor objects.
M1 = monitors.Monitor(obj=X, state_vars=['s'])
M2 = monitors.Monitor(obj=Y, state_vars=['s'])

# Add everything to the network object.
network.add_layer(layer=X, name='X')
network.add_layer(layer=Y, name='Y')
network.add_connection(connection=C, source='X', target='Y')
network.add_monitor(monitor=M1, name='X')
network.add_monitor(monitor=M2, name='Y')

# Create Poisson-distributed spike train inputs.
data = 15 * torch.rand(100)  # Generate random Poisson rates for 100 input␣
↪neurons.
train = encoding.poisson(datum=data, time=5000)  # Encode input as 5000ms Poisson␣
↪spike trains.

# Simulate network on generated spike trains.
```

(continues on next page)

(continued from previous page)

```
inputs = {'X' : train}  # Create inputs mapping.
network.run(inputs=inputs, time=5000)  # Run network simulation.

# Plot spikes of input and output layers.
spikes = {'X' : M1.get('s'), 'Y' : M2.get('s')}

fig, axes = plt.subplots(2, 1, figsize=(12, 7))
for i, layer in enumerate(spikes):
    axes[i].matshow(spikes[layer], cmap='binary')
    axes[i].set_title('%s spikes' % layer)
    axes[i].set_xlabel('Time'); axes[i].set_ylabel('Index of neuron')
    axes[i].set_xticks(()); axes[i].set_yticks(())
    axes[i].set_aspect('auto')

plt.tight_layout(); plt.show()
```

Initializes network object.

Parameters

- **dt** – Simulation timestep.
- **batch_size** – Mini-batch size.
- **learning** – Whether to allow connection updates. True by default.
- **reward_fn** – Optional class allowing for modification of reward in case of reward-modulated learning.

add_connection(*connection: bindsnet.network.topology.AbstractConnection*, *source: str*, *target: str*) → None

Adds a connection between layers of nodes to the network.

Parameters

- **connection** – An instance of class `Connection`.
- **source** – Logical name of the connection's source layer.
- **target** – Logical name of the connection's target layer.

add_layer(*layer: bindsnet.network.nodes.Nodes*, *name: str*) → None

Adds a layer of nodes to the network.

Parameters

- **layer** – A subclass of the `Nodes` object.
- **name** – Logical name of layer.

add_monitor(*monitor: bindsnet.network.monitors.AbstractMonitor*, *name: str*) → None

Adds a monitor on a network object to the network.

Parameters

- **monitor** – An instance of class `Monitor`.
- **name** – Logical name of monitor object.

clone() → bindsnet.network.network.Network

Returns a cloned network object.

Returns A copy of this network.

reset_state_variables() → None
Reset state variables of objects in network.

run(*inputs: Dict[str, torch.Tensor], time: int, one_step=False, **kwargs*) → None
Simulate network for given inputs and time.

Parameters

- **inputs** – Dictionary of `Tensor``s of shape ``[time, *input_shape]` or `[time, batch_size, *input_shape]`.
- **time** – Simulation time.
- **one_step** – Whether to run the network in "feed-forward" mode, where inputs propagate all the way through the network in a single simulation time step. Layers are updated in the order they are added to the network.

Keyword arguments:

Parameters

- **torch.Tensor] clamp** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be clamped to spiking. The `Tensor``s have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] unclamp** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be clamped to not spiking. The `Tensor``s should have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] injects_v** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be added voltage. The `Tensor``s should have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] reward** (*Union[float,*) – Scalar value used in reward-modulated learning.
- **torch.Tensor] masks** (*Dict[Tuple[str],*) – Mapping of connection names to boolean masks determining which weights to clamp to zero.
- **progress_bar** (*Bool*) – Show a progress bar while running the network.

Example:

```
import torch
import matplotlib.pyplot as plt

from bindsnet.network import Network
from bindsnet.network.nodes import Input
from bindsnet.network.monitors import Monitor

# Build simple network.
network = Network()
network.add_layer(Input(500), name='I')
network.add_monitor(Monitor(network.layers['I'], state_vars=['s']), 'I')

# Generate spikes by running Bernoulli trials on Uniform(0, 0.5) samples.
spikes = torch.bernoulli(0.5 * torch.rand(500, 500))

# Run network simulation.
network.run(inputs={'I' : spikes}, time=500)

# Look at input spiking activity.
```

(continues on next page)

(continued from previous page)

```
spikes = network.monitors['I'].get('s')
plt.matshow(spikes, cmap='binary')
plt.xticks(()); plt.yticks(());
plt.xlabel('Time'); plt.ylabel('Neuron index')
plt.title('Input spiking')
plt.show()
```

save (*file_name: str*) → None
Serializes the network object to disk.

Parameters **file_name** – Path to store serialized network object on disk.

Example:

```
import torch
import matplotlib.pyplot as plt

from pathlib           import Path
from bindsnet.network import *
from bindsnet.network import topology

# Build simple network.
network = Network(dt=1.0)

X = nodes.Input(100)  # Input layer.
Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  #␣
↪Connection from X to Y.

# Add everything to the network object.
network.add_layer(layer=X, name='X')
network.add_layer(layer=Y, name='Y')
network.add_connection(connection=C, source='X', target='Y')

# Save the network to disk.
network.save(str(Path.home()) + '/network.pt')
```

train (*mode: bool = True*) → torch.nn.modules.module.Module
Sets the node in training mode.

Parameters **mode** – Turn training on or off.

Returns `self` as specified in `torch.nn.Module`.

bindsnet.network.network.**load** (*file_name: str*, *map_location: str = 'cpu'*, *learning: bool = None*) → bindsnet.network.network.Network
Loads serialized network object from disk.

Parameters

- **file_name** – Path to serialized network object on disk.
- **map_location** – One of `"cpu"` or `"cuda"`. Defaults to `"cpu"`.
- **learning** – Whether to load with learning enabled. Default loads value from disk.

bindsnet.network.nodes module

class bindsnet.network.nodes.**AbstractInput**
Bases: abc.ABC

Abstract base class for groups of input neurons.

class bindsnet.network.nodes.**AdaptiveLIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *rest: Union[float, torch.Tensor] = -65.0*, *reset: Union[float, torch.Tensor] = -65.0*, *thresh: Union[float, torch.Tensor] = -52.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 100.0*, *theta_plus: Union[float, torch.Tensor] = 0.05*, *tc_theta_decay: Union[float, torch.Tensor] = 10000000.0*, *lbound: float = None*, ***kwargs*)
Bases: *bindsnet.network.nodes.Nodes*

Layer of leaky integrate-and-fire (LIF) neurons with adaptive thresholds. A neuron's voltage threshold is increased by some constant each time it spikes; otherwise, it is decaying back to its default value.

Instantiates a layer of LIF neurons with adaptive firing thresholds.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **rest** – Resting membrane voltage.
- **reset** – Post-spike reset voltage.
- **thresh** – Spike threshold voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.
- **theta_plus** – Voltage increase of threshold after spiking.
- **tc_theta_decay** – Time constant of adaptive threshold decay.
- **lbound** – Lower bound of the voltage.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**BoostedLIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = 13.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 100.0*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Instantiates a layer of LIF neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**CSRMNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *rest: Union[float, torch.Tensor] = -65.0*, *thresh: Union[float, torch.Tensor] = -52.0*, *responseKernel: str = 'ExponentialKernel'*, *refractoryKernel: str = 'EtaKernel'*, *tau: Union[float, torch.Tensor] = 1*, *res_window_size: Union[float, torch.Tensor] = 20*, *ref_window_size: Union[float, torch.Tensor] = 10*, *reset_const: Union[float, torch.Tensor] = 50*, *tc_decay: Union[float, torch.Tensor] = 100.0*, *theta_plus: Union[float, torch.Tensor] = 0.05*, *tc_theta_decay: Union[float, torch.Tensor] = 10000000.0*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

A layer of Cumulative Spike Response Model (Gerstner and van Hemmen 1992, Gerstner et al. 1996) nodes. It accounts for a model where refractoriness and adaptation were modeled by the combined effects of the spike after potentials of several previous spikes, rather than only the most recent spike.

Instantiates a layer of Cumulative Spike Response Model nodes.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **rest** – Resting membrane voltage.
- **thresh** – Spike threshold voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.
- **theta_plus** – Voltage increase of threshold after spiking.
- **tc_theta_decay** – Time constant of adaptive threshold decay.
- **lbound** – Lower bound of the voltage.

AlphaKernel(*dt*)

AlphaKernelSLAYER(*dt*)

EtaKernel(*dt*)

ExponentialKernel(*dt*)

LaplacianKernel(*dt*)

RectangularKernel(*dt*)

TriangularKernel(*dt*)

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**CurrentLIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *rest: Union[float, torch.Tensor] = -65.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 100.0*, *tc_i_decay: Union[float, torch.Tensor] = 2.0*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of current-based leaky integrate-and-fire (LIF) neurons. Total synaptic input current is modeled as a decaying memory of input spikes multiplied by synaptic strengths.

Instantiates a layer of synaptic input current-based LIF neurons. :param n: The number of neurons in the layer. :param shape: The dimensionality of the layer. :param traces: Whether to record spike traces. :param traces_additive: Whether to record spike traces additively. :param tc_trace: Time constant of spike trace decay. :param trace_scale: Scaling factor for spike trace. :param sum_input: Whether to sum all inputs. :param thresh: Spike threshold voltage. :param rest: Resting membrane voltage. :param reset: Post-spike reset voltage. :param refrac: Refractory (non-firing) period of the neuron. :param tc_decay: Time constant of neuron voltage decay. :param tc_i_decay: Time constant of synaptic input current decay. :param lbound: Lower bound of the voltage.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**DiehlAndCookNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *rest: Union[float, torch.Tensor] = -65.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 100.0*, *theta_plus: Union[float, torch.Tensor] = 0.05*, *tc_theta_decay: Union[float, torch.Tensor] = 10000000.0*, *lbound: float = None*, *one_spike: bool = True*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of leaky integrate-and-fire (LIF) neurons with adaptive thresholds (modified for Diehl & Cook 2015 replication).

Instantiates a layer of Diehl & Cook 2015 neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **rest** – Resting membrane voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.
- **theta_plus** – Voltage increase of threshold after spiking.
- **tc_theta_decay** – Time constant of adaptive threshold decay.
- **lbound** – Lower bound of the voltage.
- **one_spike** – Whether to allow only one spike per timestep.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**IFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of integrate-and-fire (IF) neurons.

Instantiates a layer of IF neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **lbound** – Lower bound of the voltage.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**Input**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*, *bindsnet.network.nodes.AbstractInput*

Layer of nodes with user-specified spiking behavior.

Instantiates a layer of input neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record decaying spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.

forward(*x: torch.Tensor*) → None
On each simulation step, set the spikes of the population equal to the inputs.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

class bindsnet.network.nodes.**IzhikevichNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *excitatory: float = 1*, *thresh: Union[float, torch.Tensor] = 45.0*, *rest: Union[float, torch.Tensor] = -65.0*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of **'Izhikevich neurons<https://www.izhikevich.org/publications/spikes.htm>'_**.

Instantiates a layer of Izhikevich neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **excitatory** – Percent of excitatory (vs. inhibitory) neurons in the layer; in range `[0, 1]`.
- **thresh** – Spike threshold voltage.
- **rest** – Resting membrane voltage.
- **lbound** – Lower bound of the voltage.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**LIFNodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -52.0*, *rest: Union[float, torch.Tensor] = -65.0*, *reset: Union[float, torch.Tensor] = -65.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 100.0*, *lbound: float = None*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of leaky integrate-and-fire (LIF) neurons.

Instantiates a layer of LIF neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **rest** – Resting membrane voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.
- **lbound** – Lower bound of the voltage.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**McCullochPitts**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = 1.0*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of McCulloch-Pitts neurons.

Instantiates a McCulloch-Pitts layer of neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

class bindsnet.network.nodes.**Nodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *learning: bool = True*, ***kwargs*)

Bases: torch.nn.modules.module.Module

Abstract base class for groups of neurons.

Abstract base class constructor.

Parameters

- **n** – The number of neurons in the layer.

- **shape** – The dimensionality of the layer.
- **traces** – Whether to record decaying spike traces.
- **traces_additive** – Whether to record spike traces additively.
- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **learning** – Whether to be in learning or testing.

compute_decays(*dt*) → None
Abstract base class method for setting decays.

forward(*x: torch.Tensor*) → None
Abstract base class method for a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Abstract base class method for resetting state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

train(*mode: bool = True*) → bindsnet.network.nodes.Nodes
Sets the layer in training mode.

Parameters **mode** (*bool*) – Turn training on or off

Returns self as specified in *torch.nn.Module*

class bindsnet.network.nodes.**SRM0Nodes**(*n: Optional[int] = None*, *shape: Optional[Iterable[int]] = None*, *traces: bool = False*, *traces_additive: bool = False*, *tc_trace: Union[float, torch.Tensor] = 20.0*, *trace_scale: Union[float, torch.Tensor] = 1.0*, *sum_input: bool = False*, *thresh: Union[float, torch.Tensor] = -50.0*, *rest: Union[float, torch.Tensor] = -70.0*, *reset: Union[float, torch.Tensor] = -70.0*, *refrac: Union[int, torch.Tensor] = 5*, *tc_decay: Union[float, torch.Tensor] = 10.0*, *lbound: float = None*, *eps_0: Union[float, torch.Tensor] = 1.0*, *rho_0: Union[float, torch.Tensor] = 1.0*, *d_thresh: Union[float, torch.Tensor] = 5.0*, ***kwargs*)

Bases: *bindsnet.network.nodes.Nodes*

Layer of simplified spike response model (SRM0) neurons with stochastic threshold (escape noise). Adapted from (Vasilaki et al., 2009).

Instantiates a layer of SRM0 neurons.

Parameters

- **n** – The number of neurons in the layer.
- **shape** – The dimensionality of the layer.
- **traces** – Whether to record spike traces.
- **traces_additive** – Whether to record spike traces additively.

- **tc_trace** – Time constant of spike trace decay.
- **trace_scale** – Scaling factor for spike trace.
- **sum_input** – Whether to sum all inputs.
- **thresh** – Spike threshold voltage.
- **rest** – Resting membrane voltage.
- **reset** – Post-spike reset voltage.
- **refrac** – Refractory (non-firing) period of the neuron.
- **tc_decay** – Time constant of neuron voltage decay.
- **lbound** – Lower bound of the voltage.
- **eps_0** – Scaling factor for pre-synaptic spike contributions.
- **rho_0** – Stochastic intensity at threshold.
- **d_thresh** – Width of the threshold region.

compute_decays(*dt*) → None
Sets the relevant decays.

forward(*x: torch.Tensor*) → None
Runs a single simulation step.

Parameters **x** – Inputs to the layer.

reset_state_variables() → None
Resets relevant state variables.

set_batch_size(*batch_size*) → None
Sets mini-batch size. Called when layer is added to a network.

Parameters **batch_size** – Mini-batch size.

bindsnet.network.topology module

class bindsnet.network.topology.**AbstractConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: abc.ABC, torch.nn.modules.module.Module

Abstract base method for connections between Nodes.

Constructor for abstract base class for connection objects.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects. :param nu: Learning rate for both pre- and post-synaptic events. It also

accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (LearningRule) – Modifies connection parameters according to some rule.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization.

compute(*s: torch.Tensor*) → None
Compute pre-activations of downstream neurons given spikes of upstream neurons.

Parameters **s** – Incoming spikes.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

Keyword arguments:

Parameters

- **learning** (*bool*) – Whether to allow connection updates.
- **mask** (*ByteTensor*) – Boolean mask determining which weights to clamp to zero.

class bindsnet.network.topology.**Connection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies synapses between one or two populations of neurons.

Instantiates a Connection object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects. :param nu: Learning rate for both pre- and post-synaptic events. It also

  accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (LearningRule) – Modifies connection parameters according to some rule.
- **w** (*torch.Tensor*) – Strengths of synapses.
- **b** (*torch.Tensor*) – Target population bias.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using connection weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

compute_window(*s: torch.Tensor*) → torch.Tensor

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**Conv1dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: int, stride: int = 1, padding: int = 0, dilation: int = 1, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies one-dimensional convolutional synapses between one or two populations of neurons.

Instantiates a `Conv1dConnection` object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – the size of 1-D convolutional kernel.
- **stride** – stride for convolution.
- **padding** – padding for convolution.

- **dilation** – dilation for convolution. :param nu: Learning rate for both pre- and post-synaptic events. It also

  accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (LearningRule) – Modifies connection parameters according to some rule.
- **w** (*torch.Tensor*) – Strengths of synapses.
- **b** (*torch.Tensor*) – Target population bias.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute convolutional pre-activations given spikes using layer weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
Normalize weights along the first axis according to total weight per target neuron.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**Conv2dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]] = 1, padding: Union[int, Tuple[int, int]] = 0, dilation: Union[int, Tuple[int, int]] = 1, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies two-dimensional convolutional synapses between one or two populations of neurons.

Instantiates a Conv2dConnection object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – Horizontal and vertical size of convolutional kernels.
- **stride** – Horizontal and vertical stride for convolution.
- **padding** – Horizontal and vertical padding for convolution.
- **dilation** – Horizontal and vertical dilation for convolution. :param nu: Learning rate for both pre- and post-synaptic events. It also

  accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*LearningRule*) – Modifies connection parameters according to some rule.
- **w** (*torch.Tensor*) – Strengths of synapses.
- **b** (*torch.Tensor*) – Target population bias.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor

Compute convolutional pre-activations given spikes using layer weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None

Normalize weights along the first axis according to total weight per target neuron.

reset_state_variables() → None

Contains resetting logic for the connection.

update(***kwargs*) → None

Compute connection's update rule.

class bindsnet.network.topology.**Conv3dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int, int]], stride: Union[int, Tuple[int, int, int]] = 1, padding: Union[int, Tuple[int, int, int]] = 0, dilation: Union[int, Tuple[int, int, int]] = 1, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Specifies three-dimensional convolutional synapses between one or two populations of neurons.

Instantiates a `Conv3dConnection` object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – Depth-wise, horizontal, and vertical size of convolutional kernels.
- **stride** – Depth-wise, horizontal, and vertical stride for convolution.
- **padding** – Depth-wise, horizontal, and vertical padding for convolution.
- **dilation** – Depth-wise, horizontal and vertical dilation for convolution. :param nu: Learning rate for both pre- and post-synaptic events. It also

  accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (LearningRule) – Modifies connection parameters according to some rule.
- **w** (*torch.Tensor*) – Strengths of synapses.
- **b** (*torch.Tensor*) – Target population bias.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor

Compute convolutional pre-activations given spikes using layer weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
Normalize weights along the first axis according to total weight per target neuron.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**LocalConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]], n_filters: int, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Specifies a locally connected connection between one or two populations of neurons.

Instantiates a `LocalConnection2D` object. Source population should have square size

Neurons in the post-synaptic population are ordered by receptive field; that is, if there are `n_conv` neurons in each post-synaptic patch, then the first `n_conv` neurons in the post-synaptic population correspond to the first receptive field, the second `n_conv` to the second receptive field, and so on.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – Horizontal and vertical size of convolutional kernels.
- **stride** – Horizontal and vertical stride for convolution.
- **n_filters** – Number of locally connected filters per pre-synaptic region. :param nu: Learning rate for both pre- and post-synaptic events. It also

  accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.
- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **update_rule** (*LearningRule*) – Modifies connection parameters according to some rule.
- **w** (*torch.Tensor*) – Strengths of synapses.
- **b** (*torch.Tensor*) – Target population bias.
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w

- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.
- **int] input_shape** (*Tuple[int,*) – Shape of input population if it's not `[sqrt, sqrt]`.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using layer weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

Keyword arguments:

Parameters **mask** (*ByteTensor*) – Boolean mask determining which weights to clamp to zero.

class bindsnet.network.topology.**LocalConnection1D**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: int, stride: int, n_filters: int, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies a one-dimensional local connection between one or two population of neurons supporting multi-channel inputs with shape (C, H); The logic is different from the original LocalConnection implementation (where masks were used with normal dense connections).

Instantiates a 'LocalConnection1D' object. Source population can be multi-channel. Neurons in the post-synaptic population are ordered by receptive field, i.e., if there are *n_conv* neurons in each post-synaptic patch, then the first *n_conv* neurons in the post-synaptic population correspond to the first receptive field, the second `n_conv` to the second receptive field, and so on. :param source: A layer of nodes from which the connection originates. :param target: A layer of nodes to which the connection connects. :param kernel_size: size of convolutional kernels. :param stride: stride for convolution. :param n_filters: Number of locally connected filters per pre-synaptic region.

param nu Learning rate for both pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

Parameters

- **reduction** – Method for reducing parameter updates along the minibatch dimension.

- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments: :param LearningRule update_rule: Modifies connection parameters according to some rule. :param torch.Tensor w: Strengths of synapses. :param torch.Tensor b: Target population bias. :param float wmin: Minimum allowed value on the connection weights. :param float wmax: Maximum allowed value on the connection weights. :param float norm: Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using layer weights. :param s: Incoming spikes. :return: Incoming spikes multiplied by synaptic weights (with or without

decaying spike activation).

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**LocalConnection2D**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]], n_filters: int, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies a two-dimensional local connection between one or two population of neurons supporting multi-channel inputs with shape (C, H, W); The logic is different from the original LocalConnection implementation (where masks were used with normal dense connections)

Instantiates a 'LocalConnection2D' object. Source population can be multi-channel. Neurons in the post-synaptic population are ordered by receptive field, i.e., if there are *n_conv* neurons in each post-synaptic patch, then the first *n_conv* neurons in the post-synaptic population correspond to the first receptive field, the second `n_conv` to the second receptive field, and so on. :param source: A layer of nodes from which the connection originates. :param target: A layer of nodes to which the connection connects. :param kernel_size: Horizontal and vertical size of convolutional kernels. :param stride: Horizontal and vertical stride for convolution. :param n_filters: Number of locally connected filters per pre-synaptic region.

param nu Learning rate for both pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

Parameters

- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments: :param LearningRule update_rule: Modifies connection parameters according to some rule. :param torch.Tensor w: Strengths of synapses. :param torch.Tensor b: Target population bias. :param float

wmin: Minimum allowed value on the connection weights. :param float wmax: Maximum allowed value on the connection weights. :param float norm: Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using layer weights. :param s: Incoming spikes. :return: Incoming spikes multiplied by synaptic weights (with or without

decaying spike activation).

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**LocalConnection3D**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int, int]], stride: Union[int, Tuple[int, int, int]], n_filters: int, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = 0.0, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Specifies a three-dimensional local connection between one or two population of neurons supporting multi-channel inputs with shape (C, H, W, D); The logic is different from the original LocalConnection implementation (where masks were used with normal dense connections)

Instantiates a 'LocalConnection3D' object. Source population can be multi-channel. Neurons in the post-synaptic population are ordered by receptive field, i.e., if there are *n_conv* neurons in each post-synaptic patch, then the first *n_conv* neurons in the post-synaptic population correspond to the first receptive field, the second `n_conv` to the second receptive field, and so on. :param source: A layer of nodes from which the connection originates. :param target: A layer of nodes to which the connection connects. :param kernel_size: Horizontal, vertical, and depth-wise size of convolutional kernels. :param stride: Horizontal, vertical, and depth-wise stride for convolution. :param n_filters: Number of locally connected filters per pre-synaptic region.

param nu Learning rate for both pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

Parameters

- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments: :param LearningRule update_rule: Modifies connection parameters according to some rule. :param torch.Tensor w: Strengths of synapses. :param torch.Tensor b: Target population bias. :param float wmin: Minimum allowed value on the connection weights. :param float wmax: Maximum allowed value on the connection weights. :param float norm: Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using layer weights. :param s: Incoming spikes. :return: Incoming spikes multiplied by synaptic weights (with or without

decaying spike activation).

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**MaxPoo3dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int, int]], stride: Union[int, Tuple[int, int, int]] = 1, padding: Union[int, Tuple[int, int, int]] = 0, dilation: Union[int, Tuple[int, int, int]] = 1, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies max-pooling synapses between one or two populations of neurons by keeping online estimates of maximally firing neurons.

Instantiates a `MaxPool3dConnection` object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – Depth-wise, horizontal and vertical size of convolutional kernels.
- **stride** – Depth-wise, horizontal and vertical stride for convolution.
- **padding** – Depth-wise, horizontal and vertical padding for convolution.
- **dilation** – Depth-wise, horizontal and vertical dilation for convolution.

Keyword arguments:

Parameters **decay** – Decay rate of online estimates of average firing activity.

compute(*s: torch.Tensor*) → torch.Tensor
Compute max-pool pre-activations given spikes using online firing rate estimates.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
No weights -> no normalization.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**MaxPool1dConnection**(*source: bindsnet.network.nodes.Nodes*, *target: bindsnet.network.nodes.Nodes*, *kernel_size: int*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, ***kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Specifies max-pooling synapses between one or two populations of neurons by keeping online estimates of maximally firing neurons.

Instantiates a `MaxPool1dConnection` object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – the size of 1-D convolutional kernel.
- **stride** – stride for convolution.
- **padding** – padding for convolution.
- **dilation** – dilation for convolution.

Keyword arguments:

Parameters **decay** – Decay rate of online estimates of average firing activity.

compute(*s: torch.Tensor*) → torch.Tensor
Compute max-pool pre-activations given spikes using online firing rate estimates.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
No weights -> no normalization.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**MaxPool2dConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]] = 1, padding: Union[int, Tuple[int, int]] = 0, dilation: Union[int, Tuple[int, int]] = 1, **kwargs*)

Bases: *bindsnet.network.topology.AbstractConnection*

Specifies max-pooling synapses between one or two populations of neurons by keeping online estimates of maximally firing neurons.

Instantiates a `MaxPool2dConnection` object.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects.
- **kernel_size** – Horizontal and vertical size of convolutional kernels.
- **stride** – Horizontal and vertical stride for convolution.
- **padding** – Horizontal and vertical padding for convolution.
- **dilation** – Horizontal and vertical dilation for convolution.

Keyword arguments:

Parameters **decay** – Decay rate of online estimates of average firing activity.

compute(*s: torch.Tensor*) → torch.Tensor
Compute max-pool pre-activations given spikes using online firing rate estimates.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
No weights -> no normalization.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**MeanFieldConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, weight_decay: float = 0.0, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

A connection between one or two populations of neurons which computes a summary of the pre-synaptic population to use as weighted input to the post-synaptic population.

Instantiates a `MeanFieldConnection` object. :param source: A layer of nodes from which the connection originates. :param target: A layer of nodes to which the connection connects.

param nu Learning rate for both pre- and post-synaptic events. It also accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

Parameters **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments: :param LearningRule update_rule: Modifies connection parameters according to some rule.

Parameters

- **torch.Tensor] w** (*Union[float,*) – Strengths of synapses. Can be single value or tensor of size `target`
- **torch.Tensor] wmin** (*Union[float,*) – Minimum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **torch.Tensor] wmax** (*Union[float,*) – Maximum allowed value(s) on the connection weights. Single value, or tensor of same size as w
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute pre-activations given spikes using layer weights. :param s: Incoming spikes. :return: Incoming spikes multiplied by synaptic weights (with or without

decaying spike activation).

normalize() → None
Normalize weights so each target neuron has sum of connection weights equal to `self.norm`.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

class bindsnet.network.topology.**SparseConnection**(*source: bindsnet.network.nodes.Nodes, target: bindsnet.network.nodes.Nodes, nu: Union[float, Sequence[float], Sequence[torch.Tensor], None] = None, reduction: Optional[callable] = None, weight_decay: float = None, **kwargs*)
Bases: *bindsnet.network.topology.AbstractConnection*

Specifies sparse synapses between one or two populations of neurons.

Instantiates a `Connection` object with sparse weights.

Parameters

- **source** – A layer of nodes from which the connection originates.
- **target** – A layer of nodes to which the connection connects. :param nu: Learning rate for both pre- and post-synaptic events. It also

accepts a pair of tensors to individualize learning rates of each neuron. In this case, their shape should be the same size as the connection weights.

- **reduction** – Method for reducing parameter updates along the minibatch dimension.
- **weight_decay** – Constant multiple to decay weights by on each iteration.

Keyword arguments:

Parameters

- **w** (*torch.Tensor*) – Strengths of synapses. Must be in `torch.sparse` format
- **sparsity** (*float*) – Fraction of sparse connections to use.
- **update_rule** (`LearningRule`) – Modifies connection parameters according to some rule.

- **wmin** (*float*) – Minimum allowed value on the connection weights.
- **wmax** (*float*) – Maximum allowed value on the connection weights.
- **norm** (*float*) – Total weight per target neuron normalization constant.

compute(*s: torch.Tensor*) → torch.Tensor
Compute convolutional pre-activations given spikes using layer weights.

Parameters **s** – Incoming spikes.

Returns Incoming spikes multiplied by synaptic weights (with or without decaying spike activation).

normalize() → None
Normalize weights along the first axis according to total weight per target neuron.

reset_state_variables() → None
Contains resetting logic for the connection.

update(***kwargs*) → None
Compute connection's update rule.

Module contents

class bindsnet.network.**Network**(*dt: float = 1.0, batch_size: int = 1, learning: bool = True, reward_fn: Optional[Type[bindsnet.learning.reward.AbstractReward]] = None*)
Bases: torch.nn.modules.module.Module

Central object of the bindsnet package. Responsible for the simulation and interaction of nodes and connections.

Example:

```
import torch
import matplotlib.pyplot as plt

from bindsnet          import encoding
from bindsnet.network import Network, nodes, topology, monitors

network = Network(dt=1.0)  # Instantiates network.

X = nodes.Input(100)  # Input layer.
Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  # Connection␣
↪from X to Y.

# Spike monitor objects.
M1 = monitors.Monitor(obj=X, state_vars=['s'])
M2 = monitors.Monitor(obj=Y, state_vars=['s'])

# Add everything to the network object.
network.add_layer(layer=X, name='X')
network.add_layer(layer=Y, name='Y')
network.add_connection(connection=C, source='X', target='Y')
network.add_monitor(monitor=M1, name='X')
network.add_monitor(monitor=M2, name='Y')
```

(continues on next page)

(continued from previous page)

```
# Create Poisson-distributed spike train inputs.
data = 15 * torch.rand(100)  # Generate random Poisson rates for 100 input␣
↪neurons.
train = encoding.poisson(datum=data, time=5000)  # Encode input as 5000ms Poisson␣
↪spike trains.

# Simulate network on generated spike trains.
inputs = {'X' : train}  # Create inputs mapping.
network.run(inputs=inputs, time=5000)  # Run network simulation.

# Plot spikes of input and output layers.
spikes = {'X' : M1.get('s'), 'Y' : M2.get('s')}

fig, axes = plt.subplots(2, 1, figsize=(12, 7))
for i, layer in enumerate(spikes):
    axes[i].matshow(spikes[layer], cmap='binary')
    axes[i].set_title('%s spikes' % layer)
    axes[i].set_xlabel('Time'); axes[i].set_ylabel('Index of neuron')
    axes[i].set_xticks(()); axes[i].set_yticks(())
    axes[i].set_aspect('auto')

plt.tight_layout(); plt.show()
```

Initializes network object.

Parameters

- **dt** – Simulation timestep.
- **batch_size** – Mini-batch size.
- **learning** – Whether to allow connection updates. True by default.
- **reward_fn** – Optional class allowing for modification of reward in case of reward-modulated learning.

add_connection(*connection: bindsnet.network.topology.AbstractConnection*, *source: str*, *target: str*) → None

Adds a connection between layers of nodes to the network.

Parameters

- **connection** – An instance of class `Connection`.
- **source** – Logical name of the connection's source layer.
- **target** – Logical name of the connection's target layer.

add_layer(*layer: bindsnet.network.nodes.Nodes*, *name: str*) → None

Adds a layer of nodes to the network.

Parameters

- **layer** – A subclass of the `Nodes` object.
- **name** – Logical name of layer.

add_monitor(*monitor: bindsnet.network.monitors.AbstractMonitor*, *name: str*) → None

Adds a monitor on a network object to the network.

Parameters

- **monitor** – An instance of class `Monitor`.

- **name** – Logical name of monitor object.

clone() → bindsnet.network.network.Network
Returns a cloned network object.

Returns A copy of this network.

reset_state_variables() → None
Reset state variables of objects in network.

run(*inputs: Dict[str, torch.Tensor], time: int, one_step=False, **kwargs*) → None
Simulate network for given inputs and time.

Parameters

- **inputs** – Dictionary of `Tensor``s of shape ``[time, *input_shape]` or `[time, batch_size, *input_shape]`.
- **time** – Simulation time.
- **one_step** – Whether to run the network in "feed-forward" mode, where inputs propagate all the way through the network in a single simulation time step. Layers are updated in the order they are added to the network.

Keyword arguments:

Parameters

- **torch.Tensor] clamp** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be clamped to spiking. The `Tensor``s have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] unclamp** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be clamped to not spiking. The `Tensor``s should have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] injects_v** (*Dict[str,*) – Mapping of layer names to boolean masks if neurons should be added voltage. The `Tensor``s should have shape ``[n_neurons]` or `[time, n_neurons]`.
- **torch.Tensor] reward** (*Union[float,*) – Scalar value used in reward-modulated learning.
- **torch.Tensor] masks** (*Dict[Tuple[str],*) – Mapping of connection names to boolean masks determining which weights to clamp to zero.
- **progress_bar** (*Bool*) – Show a progress bar while running the network.

Example:

```
import torch
import matplotlib.pyplot as plt

from bindsnet.network import Network
from bindsnet.network.nodes import Input
from bindsnet.network.monitors import Monitor

# Build simple network.
network = Network()
network.add_layer(Input(500), name='I')
network.add_monitor(Monitor(network.layers['I'], state_vars=['s']), 'I')

# Generate spikes by running Bernoulli trials on Uniform(0, 0.5) samples.
```

(continues on next page)

(continued from previous page)

```
spikes = torch.bernoulli(0.5 * torch.rand(500, 500))

# Run network simulation.
network.run(inputs={'I' : spikes}, time=500)

# Look at input spiking activity.
spikes = network.monitors['I'].get('s')
plt.matshow(spikes, cmap='binary')
plt.xticks(()); plt.yticks(());
plt.xlabel('Time'); plt.ylabel('Neuron index')
plt.title('Input spiking')
plt.show()
```

save (*file_name: str*) → None
Serializes the network object to disk.

Parameters **file_name** – Path to store serialized network object on disk.

Example:

```
import torch
import matplotlib.pyplot as plt

from pathlib          import Path
from bindsnet.network import *
from bindsnet.network import topology

# Build simple network.
network = Network(dt=1.0)

X = nodes.Input(100)  # Input layer.
Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  #␣
↪Connection from X to Y.

# Add everything to the network object.
network.add_layer(layer=X, name='X')
network.add_layer(layer=Y, name='Y')
network.add_connection(connection=C, source='X', target='Y')

# Save the network to disk.
network.save(str(Path.home()) + '/network.pt')
```

train (*mode: bool = True*) → torch.nn.modules.module.Module
Sets the node in training mode.

Parameters **mode** – Turn training on or off.

Returns `self` as specified in `torch.nn.Module`.

bindsnet.network.**load** (*file_name: str*, *map_location: str = 'cpu'*, *learning: bool = None*) → bindsnet.network.network.Network
Loads serialized network object from disk.

Parameters

- **file_name** – Path to serialized network object on disk.
- **map_location** – One of `"cpu"` or `"cuda"`. Defaults to `"cpu"`.
- **learning** – Whether to load with learning enabled. Default loads value from disk.

4.1.10 bindsnet.pipeline package

Submodules

bindsnet.pipeline.action module

bindsnet.pipeline.action.**select_first_spike**(*pipeline: bindsnet.pipeline.environment_pipeline.EnvironmentPipeline*, ***kwargs*) → int

Selects an action with have the highst spikes. In case of equal spiking select randomly

Parameters **pipeline** – EnvironmentPipeline with environment that has an integer action space and `spike_record` set.

Returns Action sampled from softmax over activity of similarly-sized output layer.

Keyword arguments:

Parameters **output** (*str*) – Name of output layer whose activity to base action selection on.

bindsnet.pipeline.action.**select_highest**(*pipeline: bindsnet.pipeline.environment_pipeline.EnvironmentPipeline*, ***kwargs*) → int

Selects an action with have the highst spikes. In case of equal spiking select randomly

Parameters **pipeline** – EnvironmentPipeline with environment that has an integer action space and `spike_record` set.

Returns Action sampled from softmax over activity of similarly-sized output layer.

Keyword arguments:

Parameters **output** (*str*) – Name of output layer whose activity to base action selection on.

bindsnet.pipeline.action.**select_multinomial**(*pipeline: bindsnet.pipeline.environment_pipeline.EnvironmentPipeline*, ***kwargs*) → int

Selects an action probabilistically based on spiking activity from a network layer.

Parameters **pipeline** – EnvironmentPipeline with environment that has an integer action space.

Returns Action sampled from multinomial over activity of similarly-sized output layer.

Keyword arguments:

Parameters **output** (*str*) – Name of output layer whose activity to base action selection on.

bindsnet.pipeline.action.**select_random**(*pipeline: bindsnet.pipeline.environment_pipeline.EnvironmentPipeline*, ***kwargs*) → int

Selects an action randomly from the action space.

Parameters **pipeline** – EnvironmentPipeline with environment that has an integer action space.

Returns Action randomly sampled over size of pipeline's action space.

bindsnet.pipeline.action.**select_softmax**(*pipeline: bindsnet.pipeline.environment_pipeline.EnvironmentPipeline*, ***kwargs*) → int

Selects an action using softmax function based on spiking from a network layer.

Parameters **pipeline** – EnvironmentPipeline with environment that has an integer action space and `spike_record` set.

Returns Action sampled from softmax over activity of similarly-sized output layer.

Keyword arguments:

Parameters **output** (*str*) – Name of output layer whose activity to base action selection on.

bindsnet.pipeline.base_pipeline module

class bindsnet.pipeline.base_pipeline.**BasePipeline**(*network: bindsnet.network.network.Network*, ***kwargs*)

Bases: object

A generic pipeline that handles high level functionality.

Initializes the pipeline.

Parameters **network** – Arbitrary network object, will be managed by the BasePipeline class.

Keyword arguments:

Parameters

- **save_interval** (*int*) – How often to save the network to disk.
- **save_dir** (*str*) – Directory to save network object to.
- **Any] plot_config** (*Dict[str,*) – Dict containing the plot configuration. Includes length, type ("color" or "line"), and interval per plot type.
- **print_interval** (*int*) – Interval to print text output.
- **allow_gpu** (*bool*) – Allows automatic transfer to the GPU.

get_spike_data() → Dict[str, torch.Tensor]
Get the spike data from all layers in the pipeline's network.

Returns A dictionary containing all spike monitors from the network.

get_voltage_data() → Tuple[Dict[str, torch.Tensor], Dict[str, torch.Tensor]]
Get the voltage data and threshold value from all applicable layers in the pipeline's network.

Returns Two dictionaries containing the voltage data and threshold values from the network.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the BasePipeline.

plots(*batch: Any*, *step_out: Any*) → None
Create any plots and logs for a step given the input batch and step output.

Parameters

- **batch** – The current batch. This could be anything as long as the subclass agrees upon the format in some way.
- **step_out** – The output from the step_() method.

reset_state_variables() → None
Reset the pipeline.

step(*batch: Any*, ***kwargs*) → Any
Single step of any pipeline at a high level.

Parameters **batch** – A batch of inputs to be handed to the step_() function. Standard in subclasses of BasePipeline.

Returns The output from the subclass's step_() method, which could be anything. Passed to plotting to accommodate this.

step_(*batch: Any*, ***kwargs*) → Any
Perform a pass of the network given the input batch.

Parameters **batch** – The current batch. This could be anything as long as the subclass agrees upon the format in some way.

Returns Any output that is need for recording purposes.

test() → None
A fully self contained test function.

train() → None
A fully self-contained training loop.

bindsnet.pipeline.base_pipeline.**recursive_to**(*item*, *device*)
Recursively transfers everything contained in item to the target device.

Parameters

- **item** – An individual tensor or container of tensors.
- **device** – `torch.device` pointing to `"cuda"` or `"cpu"`.

Returns A version of the item that has been sent to a device.

bindsnet.pipeline.dataloader_pipeline module

class bindsnet.pipeline.dataloader_pipeline.**DataLoaderPipeline**(*network: bindsnet.network.network.Network*, *train_ds: torch.utils.data.dataset.Dataset*, *test_ds: Optional[torch.utils.data.dataset.Dataset] = None*, ***kwargs*)

Bases: *bindsnet.pipeline.base_pipeline.BasePipeline*

A generic `DataLoader` pipeline that leverages the `torch.utils.data` setup. This still needs to be subclassed for specific implementations for functions given the dataset that will be used. An example can be seen in `TorchVisionDatasetPipeline`.

Initializes the pipeline.

Parameters

- **network** – Arbitrary `network` object.
- **train_ds** – Arbitrary `torch.utils.data.Dataset` object.
- **test_ds** – Arbitrary `torch.utils.data.Dataset` object.

test() → None
A fully self contained test function.

train() → None
Training loop that runs for the set number of epochs and creates a new `DataLoader` at each epoch.

class bindsnet.pipeline.dataloader_pipeline.**TorchVisionDatasetPipeline**(*network: bindsnet.network.network.Networ train_ds: torch.utils.data.dataset.Datas pipeline_analyzer: Optional[bindsnet.analysis.pipe = None*, ***kwargs*)

Bases: *bindsnet.pipeline.dataloader_pipeline.DataLoaderPipeline*

An example implementation of `DataLoaderPipeline` that runs all of the datasets inside of `bindsnet.datasets` that inherit from an instance of a `torchvision.datasets`. These are documented in `bindsnet/datasets/README.md`. This specific class just runs an unsupervised network.

Initializes the pipeline.

Parameters

- **network** – Arbitrary `network` object.
- **train_ds** – A `torchvision.datasets` wrapper dataset from `bindsnet.datasets`.

Keyword arguments:

Parameters **input_layer** (*str*) – Layer of the network that receives input.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the `BasePipeline`.

plots(*batch: Dict[str, torch.Tensor], *args*) → None
Create any plots and logs for a step given the input batch.

Parameters **batch** – A dictionary of the current batch. Includes image, label and encoded versions.

step_(*batch: Dict[str, torch.Tensor], **kwargs*) → None
Perform a pass of the network given the input batch. Unsupervised training (implying everything is stored inside of the `network` object, therefore returns `None`.

Parameters **batch** – A dictionary of the current batch. Includes image, label and encoded versions.

test_step()

bindsnet.pipeline.environment_pipeline module

class bindsnet.pipeline.environment_pipeline.**EnvironmentPipeline**(*network: bindsnet.network.network.Network*, *environment: bindsnet.environment.environment.Environ*
action_function: Optional[Callable] = None, *encoding: Optional[Callable] = None*, ***kwargs*)

Bases: *bindsnet.pipeline.base_pipeline.BasePipeline*

Abstracts the interaction between `Network`, `Environment`, and environment feedback action.

Initializes the pipeline.

Parameters

- **network** – Arbitrary network object.
- **environment** – Arbitrary environment.
- **action_function** – Function to convert network outputs into environment inputs.
- **encoding** – Function to encoding input.

Keyword arguments:

Parameters

- **device** (*str*) – PyTorch computing device
- **encode_factor** – coefficient for the input before encoding.
- **num_episodes** (*int*) – Number of episodes to train for. Defaults to 100.
- **output** (*str*) – String name of the layer from which to take output.
- **render_interval** (*int*) – Interval to render the environment.
- **reward_delay** (*int*) – How many iterations to delay delivery of reward.
- **time** (*int*) – Time for which to run the network. Defaults to the network's
- **overlay_input** (*int*) – Overlay the last X previous input
- **percent_of_random_action** (*float*) – chance to choose random action
- **random_action_after** (*int*) – take random action if same output action counter reach timestep.

env_step() → Tuple[torch.Tensor, float, bool, Dict[KT, VT]]

Single step of the environment which includes rendering, getting and performing the action, and accumulating/delaying rewards.

Returns An OpenAI `gym` compatible tuple with modified reward and info.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the `BasePipeline`.

plots(*gym_batch: Tuple[torch.Tensor, float, bool, Dict[KT, VT]], *args*) → None
Plot the encoded input, layer spikes, and layer voltages.

Parameters **gym_batch** – An OpenAI `gym` compatible tuple.

reset_state_variables() → None
Reset the pipeline.

step_(*gym_batch: Tuple[torch.Tensor, float, bool, Dict[KT, VT]], **kwargs*) → None
Run a single iteration of the network and update it and the reward list when done.

Parameters **gym_batch** – An OpenAI `gym` compatible tuple.

train(***kwargs*) → None
Trains for the specified number of episodes. Each episode can be of arbitrary length.

Module contents

class bindsnet.pipeline.**EnvironmentPipeline**(*network: bindsnet.network.network.Network*, *environment: bindsnet.environment.environment.Environment*, *action_function: Optional[Callable] = None*, *encoding: Optional[Callable] = None*, ***kwargs*)
Bases: *bindsnet.pipeline.base_pipeline.BasePipeline*

Abstracts the interaction between `Network`, `Environment`, and environment feedback action.

Initializes the pipeline.

Parameters

- **network** – Arbitrary network object.
- **environment** – Arbitrary environment.
- **action_function** – Function to convert network outputs into environment inputs.
- **encoding** – Function to encoding input.

Keyword arguments:

Parameters

- **device** (*str*) – PyTorch computing device
- **encode_factor** – coefficient for the input before encoding.
- **num_episodes** (*int*) – Number of episodes to train for. Defaults to 100.
- **output** (*str*) – String name of the layer from which to take output.
- **render_interval** (*int*) – Interval to render the environment.
- **reward_delay** (*int*) – How many iterations to delay delivery of reward.
- **time** (*int*) – Time for which to run the network. Defaults to the network's
- **overlay_input** (*int*) – Overlay the last X previous input
- **percent_of_random_action** (*float*) – chance to choose random action

- **random_action_after** (*int*) – take random action if same output action counter reach timestep.

env_step() → Tuple[torch.Tensor, float, bool, Dict[KT, VT]]
Single step of the environment which includes rendering, getting and performing the action, and accumulating/delaying rewards.

Returns An OpenAI `gym` compatible tuple with modified reward and info.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the `BasePipeline`.

plots(*gym_batch: Tuple[torch.Tensor, float, bool, Dict[KT, VT]], *args*) → None
Plot the encoded input, layer spikes, and layer voltages.

Parameters **gym_batch** – An OpenAI `gym` compatible tuple.

reset_state_variables() → None
Reset the pipeline.

step_(*gym_batch: Tuple[torch.Tensor, float, bool, Dict[KT, VT]], **kwargs*) → None
Run a single iteration of the network and update it and the reward list when done.

Parameters **gym_batch** – An OpenAI `gym` compatible tuple.

train(***kwargs*) → None
Trains for the specified number of episodes. Each episode can be of arbitrary length.

class bindsnet.pipeline.**BasePipeline**(*network: bindsnet.network.network.Network*, ***kwargs*)

Bases: `object`

A generic pipeline that handles high level functionality.

Initializes the pipeline.

Parameters **network** – Arbitrary network object, will be managed by the `BasePipeline` class.

Keyword arguments:

Parameters

- **save_interval** (*int*) – How often to save the network to disk.
- **save_dir** (*str*) – Directory to save network object to.
- **Any] plot_config** (*Dict[str,*) – Dict containing the plot configuration. Includes length, type (`"color"` or `"line"`), and interval per plot type.
- **print_interval** (*int*) – Interval to print text output.
- **allow_gpu** (*bool*) – Allows automatic transfer to the GPU.

get_spike_data() → Dict[str, torch.Tensor]
Get the spike data from all layers in the pipeline's network.

Returns A dictionary containing all spike monitors from the network.

get_voltage_data() → Tuple[Dict[str, torch.Tensor], Dict[str, torch.Tensor]]
Get the voltage data and threshold value from all applicable layers in the pipeline's network.

Returns Two dictionaries containing the voltage data and threshold values from the network.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the BasePipeline.

plots(*batch: Any*, *step_out: Any*) → None
Create any plots and logs for a step given the input batch and step output.

Parameters

- **batch** – The current batch. This could be anything as long as the subclass agrees upon the format in some way.
- **step_out** – The output from the step_() method.

reset_state_variables() → None
Reset the pipeline.

step(*batch: Any*, ***kwargs*) → Any
Single step of any pipeline at a high level.

Parameters **batch** – A batch of inputs to be handed to the step_() function. Standard in subclasses of BasePipeline.

Returns The output from the subclass's step_() method, which could be anything. Passed to plotting to accommodate this.

step_(*batch: Any*, ***kwargs*) → Any
Perform a pass of the network given the input batch.

Parameters **batch** – The current batch. This could be anything as long as the subclass agrees upon the format in some way.

Returns Any output that is need for recording purposes.

test() → None
A fully self contained test function.

train() → None
A fully self-contained training loop.

class bindsnet.pipeline.**DataLoaderPipeline**(*network: bindsnet.network.network.Network*, *train_ds: torch.utils.data.dataset.Dataset*, *test_ds: Optional[torch.utils.data.dataset.Dataset] = None*, ***kwargs*)
Bases: *bindsnet.pipeline.base_pipeline.BasePipeline*

A generic DataLoader pipeline that leverages the torch.utils.data setup. This still needs to be subclassed for specific implementations for functions given the dataset that will be used. An example can be seen in TorchVisionDatasetPipeline.

Initializes the pipeline.

Parameters

- **network** – Arbitrary network object.
- **train_ds** – Arbitrary torch.utils.data.Dataset object.
- **test_ds** – Arbitrary torch.utils.data.Dataset object.

test() → None
A fully self contained test function.

train() → None
Training loop that runs for the set number of epochs and creates a new DataLoader at each epoch.

class bindsnet.pipeline.**TorchVisionDatasetPipeline**(*network: bindsnet.network.network.Network*, *train_ds: torch.utils.data.dataset.Dataset*, *pipeline_analyzer: Optional[bindsnet.analysis.pipeline_analysis.PipelineAnalyze = None*, ***kwargs*)

Bases: *bindsnet.pipeline.dataloader_pipeline.DataLoaderPipeline*

An example implementation of DataLoaderPipeline that runs all of the datasets inside of bindsnet.datasets that inherit from an instance of a torchvision.datasets. These are documented in bindsnet/datasets/README.md. This specific class just runs an unsupervised network.

Initializes the pipeline.

Parameters

- **network** – Arbitrary network object.
- **train_ds** – A torchvision.datasets wrapper dataset from bindsnet.datasets.

Keyword arguments:

Parameters **input_layer** (*str*) – Layer of the network that receives input.

init_fn() → None
Placeholder function for subclass-specific actions that need to happen during the construction of the BasePipeline.

plots(*batch: Dict[str, torch.Tensor], *args*) → None
Create any plots and logs for a step given the input batch.

Parameters **batch** – A dictionary of the current batch. Includes image, label and encoded versions.

step_(*batch: Dict[str, torch.Tensor], **kwargs*) → None
Perform a pass of the network given the input batch. Unsupervised training (implying everything is stored inside of the network object, therefore returns None.

Parameters **batch** – A dictionary of the current batch. Includes image, label and encoded versions.

test_step()

4.1.11 bindsnet.preprocessing package

Submodules

bindsnet.preprocessing.preprocessing module

class bindsnet.preprocessing.preprocessing.**AbstractPreprocessor**
Bases: abc.ABC

Abstract base class for Preprocessor.

process(*csvfile: str*, *use_cache: bool = True*, *cachedfile: str = './processed/data.pt'*) → torch._VariableFunctionsClass.tensor

Module contents

class bindsnet.preprocessing.**AbstractPreprocessor**
Bases: abc.ABC

Abstract base class for Preprocessor.

process(*csvfile: str*, *use_cache: bool = True*, *cachedfile: str = './processed/data.pt'*) → torch._VariableFunctionsClass.tensor

4.2 Submodules

4.3 bindsnet.utils module

bindsnet.utils.**col2im_indices**(*cols: torch.Tensor, x_shape: Tuple[int, int, int, int], kernel_height: int, kernel_width: int, padding: Tuple[int, int] = (0, 0), stride: Tuple[int, int] = (1, 1)*) → torch.Tensor
col2im is a special case of fold which is implemented inside of Pytorch.

Parameters

- **cols** – Image tensor in column-wise format.
- **x_shape** – Shape of original image tensor.
- **kernel_height** – Height of the convolutional kernel in pixels.
- **kernel_width** – Width of the convolutional kernel in pixels.
- **padding** – Amount of zero padding on the input image.
- **stride** – Amount to stride over image by per convolution.

Returns Image tensor in original image shape.

bindsnet.utils.**get_square_assignments**(*assignments: torch.Tensor*, *n_sqrt: int*) → torch.Tensor
Return a grid of assignments.

Parameters

- **assignments** – Vector of integers corresponding to class labels.
- **n_sqrt** – Square root of no. of assignments.

Returns Reshaped square matrix of assignments.

bindsnet.utils.**get_square_weights**(*weights: torch.Tensor, n_sqrt: int, side: Union[int, Tuple[int, int]]*) → torch.Tensor
Return a grid of a number of filters `sqrt ** 2` with side lengths `side`.

Parameters

- **weights** – Two-dimensional tensor of weights for two-dimensional data.
- **n_sqrt** – Square root of no. of filters.
- **side** – Side length(s) of filter.

Returns Reshaped weights to square matrix of filters.

bindsnet.utils.**im2col_indices**(*x: torch.Tensor*, *kernel_height: int*, *kernel_width: int*, *padding: Tuple[int, int] = (0, 0)*, *stride: Tuple[int, int] = (1, 1)*) → torch.Tensor

im2col is a special case of unfold which is implemented inside of Pytorch.

Parameters

- **x** – Input image tensor to be reshaped to column-wise format.
- **kernel_height** – Height of the convolutional kernel in pixels.
- **kernel_width** – Width of the convolutional kernel in pixels.
- **padding** – Amount of zero padding on the input image.
- **stride** – Amount to stride over image by per convolution.

Returns Input tensor reshaped to column-wise format.

bindsnet.utils.**reshape_conv2d_weights**(*weights: torch.Tensor*) → torch.Tensor

Flattens a connection weight matrix of a Conv2dConnection

Parameters

- **weights** – Weight matrix of Conv2dConnection object.
- **wmin** – Minimum allowed weight value.
- **wmax** – Maximum allowed weight value.

bindsnet.utils.**reshape_local_connection_2d_weights**(*w: torch.Tensor, n_filters: int, kernel_size: Union[int, Tuple[int, int]], conv_size: Union[int, Tuple[int, int]], input_sqrt: Union[int, Tuple[int, int]]*) → torch.Tensor

Reshape a slice of weights of a LocalConnection2D slice for plotting. :param w: Slice of weights from a LocalConnection2D object. :param n_filters: Number of filters (output channels). :param kernel_size: Side length(s) of convolutional kernel. :param conv_size: Side length(s) of convolution population. :param input_sqrt: Sides length(s) of input neurons. :return: A slice of LocalConnection2D weights reshaped as a collection of spatially ordered square grids.

bindsnet.utils.**reshape_locally_connected_weights**(*w: torch.Tensor, n_filters: int, kernel_size: Union[int, Tuple[int, int]], conv_size: Union[int, Tuple[int, int]], locations: torch.Tensor, input_sqrt: Union[int, Tuple[int, int]]*) → torch.Tensor

Get the weights from a locally connected layer and reshape them to be two-dimensional and square.

Parameters

- **w** – Weights from a locally connected layer.
- **n_filters** – No. of neuron filters.
- **kernel_size** – Side length(s) of convolutional kernel.
- **conv_size** – Side length(s) of convolution population.
- **locations** – Binary mask indicating receptive fields of convolution population neurons.
- **input_sqrt** – Sides length(s) of input neurons.

Returns Locally connected weights reshaped as a collection of spatially ordered square grids.

4.4 Module contents

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

b

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W