bindsnet Documentation

Daniel Saunders, Hananel Hazan

Jun 02, 2026

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

ONE

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

TWO

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

THREE

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).

For more complex connections, the MulticompartmentConnection class can be used. The MulticompartmentConnection will pass spikes through different "features" such as weights, bias's, and boolean masks in a specified order. Features are passed to the MulticompartmentConnection constructor in a list, and executed in order. For example, the code below uses a pipeline containing a weight and bias feature. During runtime, spikes from the source will be multiplied by the weights first, then a bias added second. Additional features can be added before/after/between these two. To create a simple all-to-all connection with a weight and bias:

```
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import MulticompartmentConnection
from bindsnet.network.topology_features import Weight, Bias

# 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)

# Create 'pipeline' of features that spikes will pass through
weights = Weight(name='weight_feature', value=torch.rand(100, 1000))
bias = Bias(name='bias_feature', value=torch.rand(100, 1000))

# Connect the two layers.
connection = MulticompartmentConnection(
    source=source_layer, target=target_layer,
    pipeline=[weight, bias]
)
```

Feature values (e.g., weights, biases) can be represented as sparse tensors for memory efficiency. To enable sparse tensor support:

1. Set the sparse=True parameter when initializing the feature.
2. If the value tensor does not include a batch dimension, explicitly specify the batch_size parameter.

Example 1: Batch dimension included in the value tensor (first axis):

```
weights = Weight(name='weight_feature', value=torch.rand(2, 100, 1000), sparse=True)  #␣
↪Batch size = 2
bias = Bias(name='bias_feature', value=torch.rand(2, 100, 1000), sparse=True)
```

Example 2: Batch dimension specified via batch_size parameter (no batch axis in value):

```
weights = Weight(name='weight_feature', value=torch.rand(100, 1000), sparse=True, batch_
↪size=2)
bias = Bias(name='bias_feature', value=torch.rand(100, 1000), sparse=True, batch_size=2)
```

Note that subtraction and addition operations for sparse tensors are 90 times slower than those for dense tensors (tested on RTX 3060)

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

network = Network()

source_layer = Input(n=100)
```

(continues on next page)

(continued from previous page)

```
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.

Monitors are not officially supported for MulticompartmentConnection

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.

`bindsnet.network.monitors.AbstractMonitor` objects can also store sparse tensor-valued variables. For example, spikes can be stored efficiently using a sparse monitor:

```
Monitor(
    network.layers[layer], state_vars=["s"], time=int(time / dt), device=device,␣
↪sparse=True
)
```

Performance Considerations:

While sparse tensors reduce memory usage when the percentage of non-zero values is below 4% (see table below), there is a trade-off in computational speed. Benchmarks on an RTX 3070 GPU show:

- Sparse runtime: 1.2 seconds
- Dense runtime: 0.5 seconds

The dense implementation achieves 2x faster execution compared to sparse tensors in this configuration.

Sparse (megabytes used)	Dense (megabytes used)	Ratio (Sparse/Dense) %	% of non zero values
15	119	13	0.5
30	119	25	1.0
45	119	38	1.5
60	119	50	2.0
75	119	63	2.5
89	119	75	3.0
104	119	87	3.5
119	119	100	4.0
134	119	113	4.5
149	119	125	5.0
164	119	138	5.5
179	119	150	6.0
194	119	163	6.5
209	119	176	7.0
224	119	188	7.5
239	119	201	8.0
253	119	213	8.5
268	119	225	9.0
283	119	238	9.5

This table and performance metrics were generated by `examples/benchmark/sparse_vs_dense_tensors.py`

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.

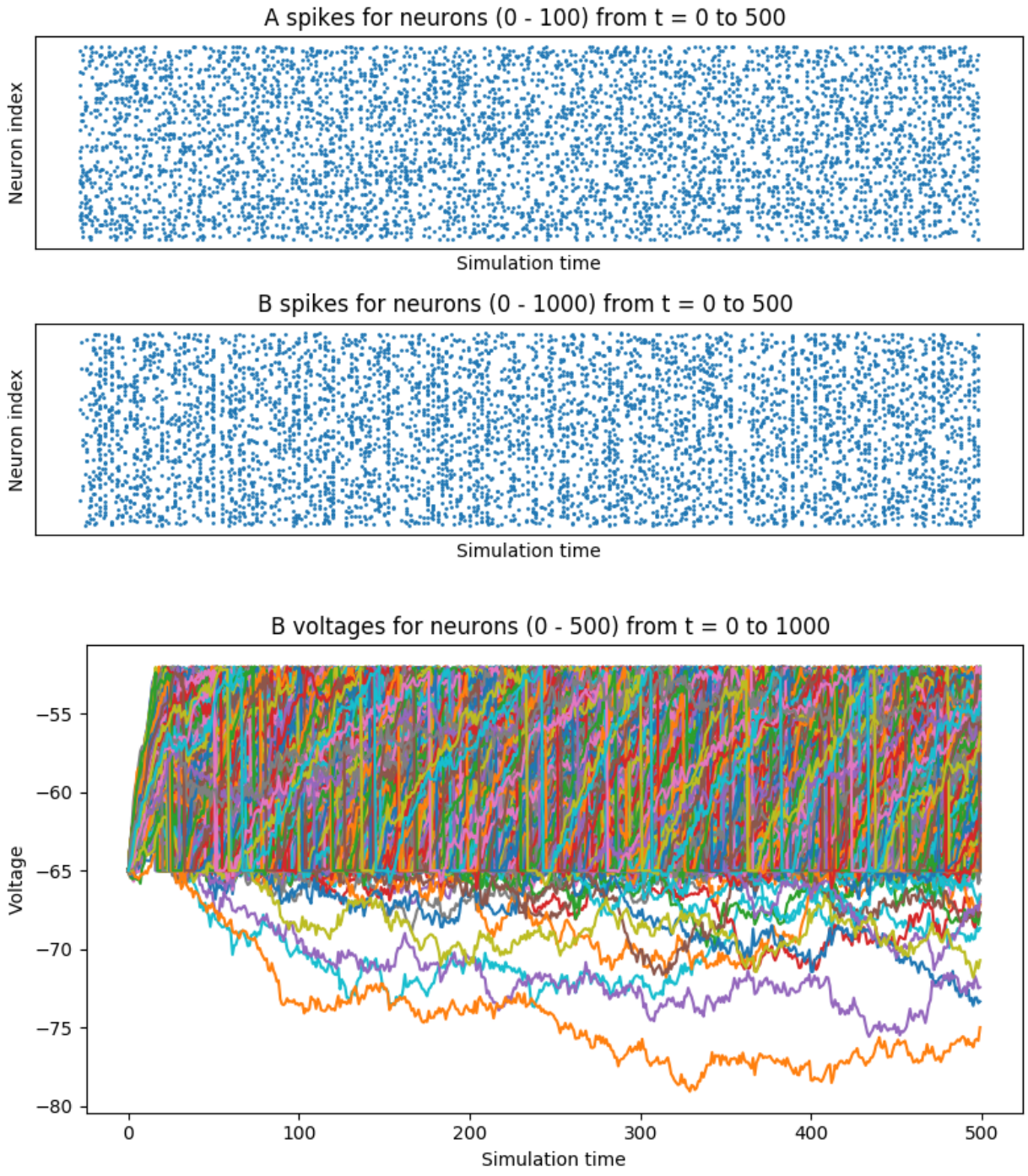
```
import torch
import matplotlib.pyplot as plt
from bindsnet.network import Network
from bindsnet.network.nodes import Input, LIFNodes
from bindsnet.network.topology import Connection
from bindsnet.network.monitors import Monitor
from bindsnet.analysis.plotting import plot_spikes, plot_voltages

# 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"
)

# Create recurrent connection in output layer.
recurrent_connection = Connection(
    source=target_layer,
    target=target_layer,
```

(continues on next page)

(continued from previous page)

```
    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.

3.1.5 Lowering precision

You can choose the precision for the weights. It can be specified as the `value_dtype` parameter of the Weight class.

```
MulticompartmentConnection(
    ...
    pipeline=[
        Weight(
            'weight',
            w,
            value_dtype='float16',
            ...
        )
    ]
)
```

Below is the performance statistics for float16 and float32.

The data was obtained by running examples/benchmark/lowering_precision.py

```
precision: float32
Time (sec) | GPU memory (Mb)
19.7812    | 52
19.4812    | 52
19.0769    | 52
19.1530    | 52
Average time: 19.373075
Average memory: 52.0

precision: float16
Time (sec) | GPU memory (Mb)
19.5023    | 49
20.5734    | 49
19.8735    | 49
19.8931    | 49
Average time: 19.960575
Average memory: 49.0
```

As you can see, reducing from float32 to float16 does not provide a significant advantage in terms of time or memory. The float16 option only reduces memory usage by 6%.

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 modulates plasticity on a more global level; e.g., for all synapses in the network. Examples include:**
  - (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

# 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)
```

(continues on next page)

(continued from previous page)

```
# 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.

If using a MulticompartmentConneciton, you can add a learning rule to a specific feature. Note that only `NoOp`, `PostPre`, `MSTDP`, `MSTDPET` are supported, and located at bindsnet.learning.MCC_learning. Below is an example of how to apply a PostPre learning rule to a weight function. Note that the bias does not have a learning rule, so it will remain static.

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

# 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)

# Create 'pipeline' of features that spikes will pass through
weights = Weight(name='weight_feature', value=torch.rand(100, 1000),
                 learning_rule=PostPre, nu=(1e-4, 1e-2))
bias = Bias(name='bias_feature', value=torch.rand(100, 1000))

# Connect the two layers.
connection = MulticompartmentConnection(
    source=source_layer, target=target_layer,
    pipeline=[weights, bias])
)
```

CHAPTER

FOUR

BINDSNET PACKAGE

4.1 Subpackages

4.1.1 bindsnet.analysis package

Submodules

bindsnet.analysis.pipeline_analysis module

bindsnet.analysis.plotting module

bindsnet.analysis.visualization module

Module contents

4.1.2 bindsnet.conversion package

Submodules

bindsnet.conversion.conversion module

bindsnet.conversion.nodes module

bindsnet.conversion.topology module

Module contents

4.1.3 bindsnet.datasets package

Submodules

bindsnet.datasets.alov300 module

bindsnet.datasets.collate module

bindsnet.datasets.dataloader module

bindsnet.datasets.davis module

bindsnet.datasets.preprocess module

bindsnet.datasets.spoken_mnist module

bindsnet.datasets.torchvision_wrapper module

Module contents

4.1.4 bindsnet.encoding package

Submodules

bindsnet.encoding.encoders module

bindsnet.encoding.encodings module

bindsnet.encoding.loaders module

Module contents

4.1.5 bindsnet.environment package

Submodules

bindsnet.environment.environment module

Module contents

4.1.6 bindsnet.evaluation package

Submodules

bindsnet.evaluation.evaluation module

Module contents

4.1.7 bindsnet.learning package

Submodules

bindsnet.learning.learning module

bindsnet.learning.reward module

Module contents

4.1.8 bindsnet.models package

Submodules

bindsnet.models.models module

Module contents

4.1.9 bindsnet.network package

Submodules

bindsnet.network.monitors module

bindsnet.network.network module

bindsnet.network.nodes module

bindsnet.network.topology module

Module contents

4.1.10 bindsnet.pipeline package

Submodules

bindsnet.pipeline.action module

bindsnet.pipeline.base_pipeline module

bindsnet.pipeline.dataloader_pipeline module

bindsnet.pipeline.environment_pipeline module

Module contents

4.1.11 bindsnet.preprocessing package

Submodules

bindsnet.preprocessing.preprocessing module

Module contents

4.2 Submodules

4.3 bindsnet.utils module

4.4 Module contents

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search