ngc-learn

Release 0.4.0

Alexander Ororbia

Jun 02, 2022

INTRODUCTION:

CHAPTER

ONE

OVERVIEW

ngc-learn is a Python library for building, simulating, and analyzing arbitrary predictive processing/coding models based on the neural generative coding (NGC) computational framework as well as other neurobiologically-motivated/grounded systems. This toolkit is built on top of Tensorflow 2 and is distributed under the 3-Clause BSD license.

Advances made in research on artificial neural networks (ANNs) have led to many breakthroughs in machine learning and beyond, resulting in the design of powerful models that can categorize and forecast as well as agents that can play games and solve complex problems. Behind these achievements is the backpropagation of errors (or backprop) algorithm. Although elegant and powerful, a major long-standing criticism of backprop has been its biological implausibility. In short, it is not likely that the brain adjusts the synapses that connect the billions of neurons that compose it in the way that backprop would prescribe.

Although ANNs are (loosely) inspired by our current understanding of the human brain, the connections to the actual mechanisms that drive systems of natural neurons are quite loose, at best. Although the question as to how the brain exactly conducts credit assignment – or the process of determining the contribution of each and every neuron to the system's overall error on some task (the "blame game") – is still an open one, it would prove invaluable to have a flexible computational and software framework that can facilitate the design and development of brain-inspired neural systems that can also learn complex tasks. These tasks range from generative modeling to interacting and manipulating dynamically-evolving environments. This would benefit researchers in fields including, but not limited to, machine learning, (computational) neuroscience, and cognitive science.

ngc-learn aims to fill the above need by concretely instantiating an important theory in neuroscience known as predictive processing, positing that the brain is largely a continual prediction engine, constantly hypothesizing the state of its environment and updating its own internal mental model of it as data is gathered. Moreover, prediction and correction happen at many levels or regions within the brain – clusters or groups of neurons in one region attempt to predict the state of neurons at another region, forming a complex, somewhat hierarchical structure that includes neurons which attempt to predict actual sensory input. Neurons within this system adjust their internal activity values (as well the strengths of the synapses that wire to them) based on how different their predictions were from observed signals. Concretely, ngc-learn implements a general predictive processing framework known as neural generative coding (NGC).

The overarching goal of ngc-learn is to provide researchers and engineers with:

- a modular design that allows for the flexible creation, simulation, and analysis of neural systems fundamentally built and driven by predictive processing;
- a powerful, approachable tool, written by and maintained by researchers and experimenters directly studying and working to advance predictive processing, meant to lower the barriers to entry to this field of research;
- a *"model museum"* that captures the essence of fundamental and interesting predictive processing models and algorithms throughout history, allowing for the study of and experimentation with classical and modern ideas.

The ngc-learn software framework was originally developed in 2019 by the Neural Adaptive Computing (NAC) laboratory in Rochester Institute of Technology meant as an internal tool for predictive processing research (with earlier incarnations in the Scala programming language, dating back to early 2017). It remains actively maintained and used for predictive processing research in NAC (see ngc-learn's mention/announcement in this engineering blog post). We warmly welcome community contributions to this project. For details please check out our contributing guidelines.

1.1 Citation

Please cite ngc-learn's source/core paper if you use this framework in your publications:

```
@article{Ororbia2022,
  author={Ororbia, Alexander and Kifer, Daniel},
  title={The neural coding framework for learning generative models},
  journal={Nature Communications},
  year={2022},
  month={Apr},
  day={19},
  volume={13},
  number={1},
  pages={2064},
  issn={2041-1723},
  doi={10.1038/s41467-022-29632-7},
  url={https://doi.org/10.1038/s41467-022-29632-7}
}
```

CHAPTER

TWO

INSTALLATION

ngc-learn officially supports Linux on Python 3. It can be run with or without a GPU.

Setup: Ensure that you have installed the following base dependencies in your system. Note that this library was developed and tested on Ubuntu 18.04. Specifically, ngc-learn requires:

- Python (>=3.7)
- Numpy (>=1.20.0)
- Tensorflow 2.0.0, specifically, tensorflow-gpu>=2.0.0
- scikit-learn (>=0.24.2) (needed for the demonstrations in `examples/` as well as ngclearn.density)
- matplotlib (>=3.4.3) (needed for the demonstrations in `examples/`)

2.1 Install from Source

1. Clone the ngc-learn repository:

```
$ git clone https://github.com/ago109/ngc-learn.git
$ cd ngc-learn
```

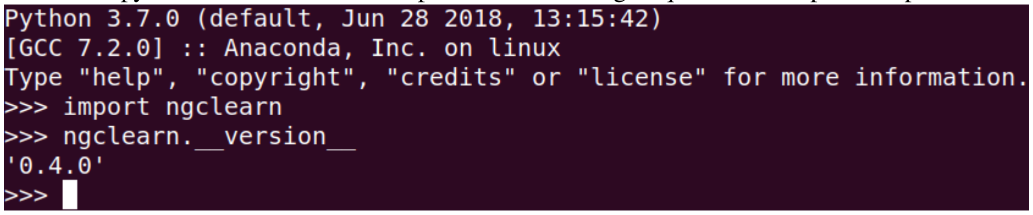
2. Install the base requirements (and a few extras for building the docs) with:

```
$ pip3 install -r requirements.txt
```

3. Install the ngc-learn package via:

```
$ python setup.py install
```

If the installation was successful, you should see the following if you test it against your Python interpreter, i.e., run the $ python command and complete the following sequence of steps as depicted in the screenshot below:

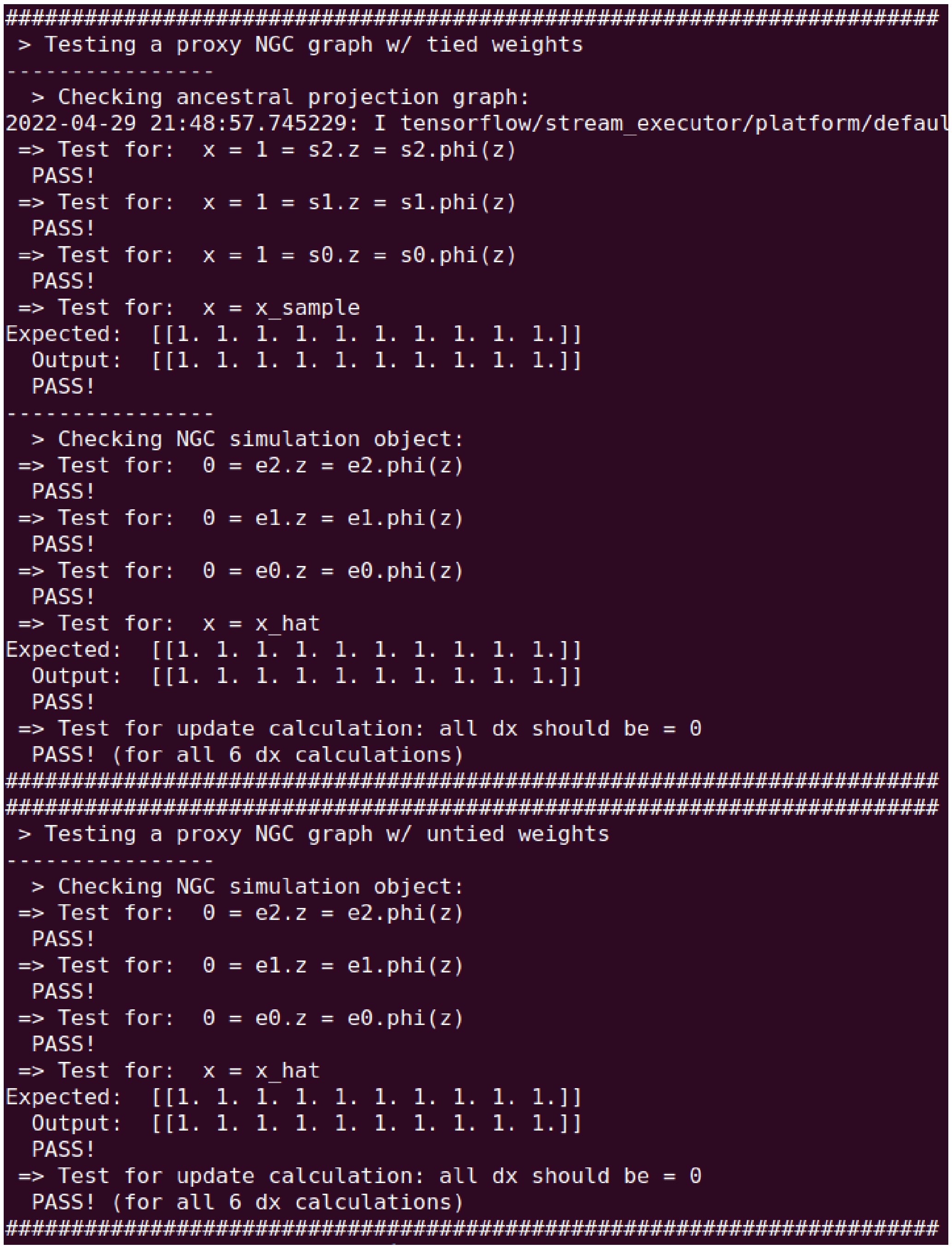


```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ngclearn
>>> ngclearn.__version__
'0.4.0'
>>> 
```

After installation, you can also the tests in the directory `/tests/`, specifically

```
$ python test_fun_dynamics.py
```

and should see that all the basic assertion tests yield pass as follows:



```
######################################################################
 > Testing a proxy NGC graph w/ tied weights
----------------
  > Checking ancestral projection graph:
2022-04-29 21:48:57.745229: I tensorflow/stream_executor/platform/defaul
 => Test for:  x = 1 = s2.z = s2.phi(z)
  PASS!
 => Test for:  x = 1 = s1.z = s1.phi(z)
  PASS!
 => Test for:  x = 1 = s0.z = s0.phi(z)
  PASS!
 => Test for:  x = x_sample
Expected:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  Output:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  PASS!
----------------
  > Checking NGC simulation object:
 => Test for:  0 = e2.z = e2.phi(z)
  PASS!
 => Test for:  0 = e1.z = e1.phi(z)
  PASS!
 => Test for:  0 = e0.z = e0.phi(z)
  PASS!
 => Test for:  x = x_hat
Expected:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  Output:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  PASS!
 => Test for update calculation: all dx should be = 0
  PASS! (for all 6 dx calculations)
######################################################################
######################################################################
 > Testing a proxy NGC graph w/ untied weights
----------------
  > Checking NGC simulation object:
 => Test for:  0 = e2.z = e2.phi(z)
  PASS!
 => Test for:  0 = e1.z = e1.phi(z)
  PASS!
 => Test for:  0 = e0.z = e0.phi(z)
  PASS!
 => Test for:  x = x_hat
Expected:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  Output:  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
  PASS!
 => Test for update calculation: all dx should be = 0
  PASS! (for all 6 dx calculations)
######################################################################
```

2.2 A Note on Simulating with the GPU or CPU

Simulations using ngc-learn can be run on either the CPU or GPU (currently, in this version of ngc-learn, there is no multi-CPU/GPU support) by writing code near the top of your general simulation scripts as follows:

```
mid = -1 # the gpu_id (run nivida-smi to find your system's GPU identifiers)
if mid >= 0:
    print(" > Using GPU ID {0}".format(mid))
    os.environ["CUDA_VISIBLE_DEVICES"]="{0}".format(mid)'
    gpu_tag = '/GPU:0'
else:
    os.environ["CUDA_VISIBLE_DEVICES"]="-1"
    gpu_tag = '/CPU:0'

...other non-initialization/simulation code goes here...

with tf.device(gpu_tag): # forces code below here to reisde on GPU with identifer "mid"
    ...initialization and simulation code goes here...
```

where `mid = -1` triggers a CPU-only simulation while `mid >= 0` would trigger a GPU simulation based on the identifier provided (an `mid = 0` would force the simulation to take place on GPU with an identifier of `0` – you can query the identifiers of what GPUs your system houses with the bash command `$ nvidia-smi`).

Note that, as shown in the code snippet above, later on in your script, before the code you write that executes things such as initializing NGC graphs or simulating the NGC systems (learning, inference, etc.), it is recommended to place a with-statement before the relevant code (which forces the execution of the following code indented underneath the with-statement to reside on the GPU with the identifier you provided.)

CHAPTER

THREE

LESSON 1: THE NODES-AND-CABLES SYSTEM

In this tutorial, we will focus on working through the very basics of ngc-learn's nodes-and-cables system. Specifically, you will learn how various (mini-)circuits are built in order to develop an intuition of how these fundamental modeling blocks fit together and how, when they are put together in the right way, you can simulate your own evolving dynamical neural systems.

We recommend that you create a directory labeled `tutorials/` and a sub-directory within labeled as `lesson1/` for you to place the code/Python scripts that you will write in throughout this lesson.

3.1 Theory: Cable Theory and Neural Compartments

At its core, part of ngc-learn's core design is inspired by (neural) cable theory , where neuronal units, which are arranged in complex connectivity structures, are viewed as performing dendritic calculations (of varying complexity). In essence, a particular neuron integrates information from different input signal sources (for example, signals produced by other neurons), in often highly nonlinear ways through a complex dendritic tree.

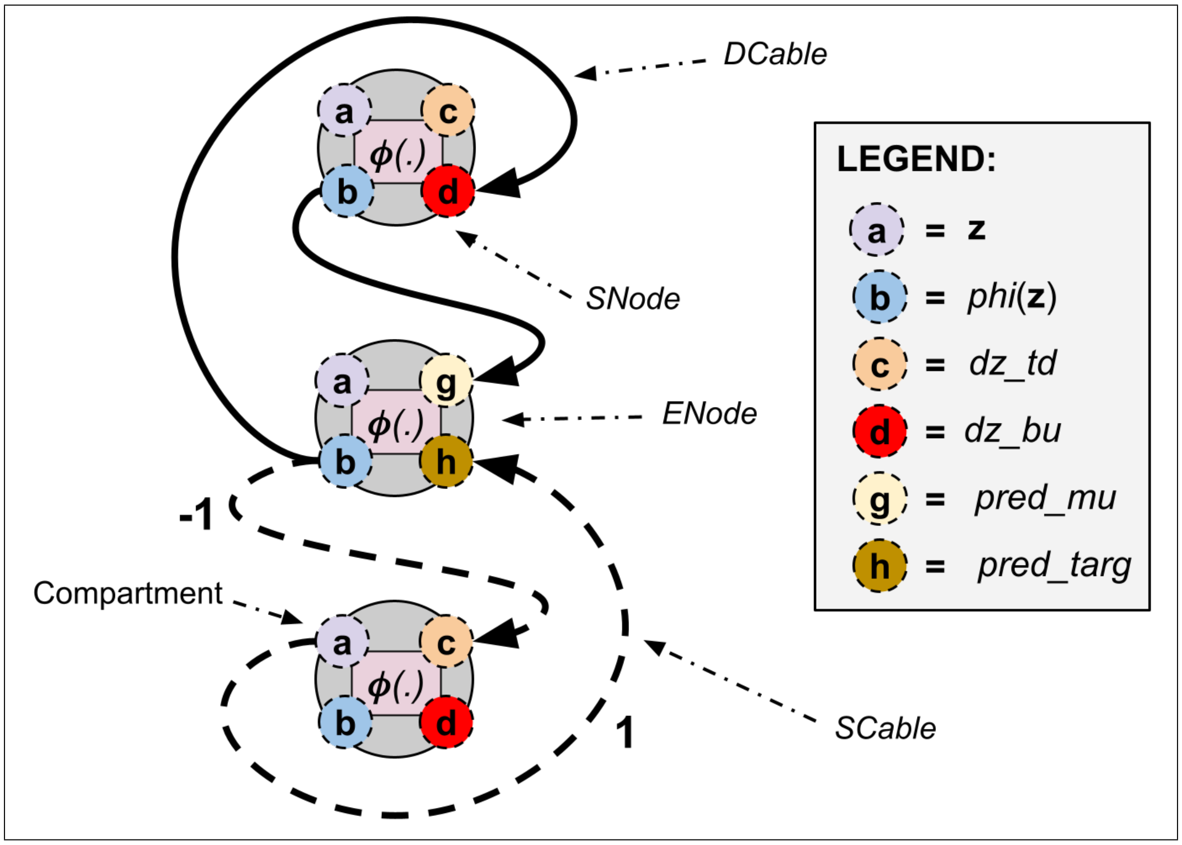
Although modeling a complete neuronal system through the lens of cable theory is complex and intricate in of itself, ngc-learn is built with this direction in mind. ngc-learn starts with with the idea a neuron (or a cluster of them) can be viewed as a node, or *Node* (also see *Node Model*), and each bundle of synapses that connect pairs of nodes can be viewed as a cable, or *Cable* (also see *Cable Model*).

Each node has multiple, different (named) "compartments", which are regions or slots within the node that other nodes can deposit information/signals into. These compartments allow a node to collect information from many different connected/related nodes and then decide how to combine these different signals in order calculate its own output activity (either in the form of a rate-coded firing rate or binary spikes) using the integration logic defined within its own specific `step()` function. When an NGC system, composed of many of these nodes, is simulated over a period of time (processing some form of sensory input), its underlying simulation object (the `NGCGraph`) calls the `step()` routine of each constituent node within one discrete time step. The order in which the node `step()` routines are called is governed by "execution cycles", which are defined by the experimenter at object initialization, for example, a user might want all of the state nodes to first execute their internal step logic before the error nodes do (which can be done by specifying two distinct cycles in the order desired).

As a result, many nodes and cables result in an NGC system where each node is itself, in general, a stateful computation even if we are processing inherently non-temporal data such as static images.

3.2 Node and Cable Fundamentals

To start creating predictive processing models and other neurobiological neural systems, we must first examine the fundamental building blocks you will need to craft them. At a high level, motivated by the theory described above, an NGC system is made up of multiple nodes and cables where each node (or cluster/group of neurons) in the system contains one or more compartments (or vectors of scalar numbers/signals) and each cable transmits the information (vector of numbers) inside one compartment within one node and transforms this information (potentially with synapses) and finally deposits this transformed information into one single compartment of another node. Understanding how nodes and cables relate to each other in ngc-learn is necessary if one wants to build and simulate their own custom NGC system (for example, the arbitrary 3-node one graphically depicted in the figure below).



3.2.1 The Node

First, let us examine the *node object* itself. A node (or `Node`) contains inside of it a cluster (or block) of neurons, the number of which is controlled through the `dim` argument. We will, in this tutorial lesson, examine two core node types within ngc-learn, the stateful node (or *SNode*) and the error node (*ENode*), although there are other node types (such as convenience nodes like the `FNode` or spiking nodes).

Every node in ngc-learn has several compartments, which are made explicit in each node's documentation listed under "Compartments:" and the names of which can be programmatically accessed through the node's data member `.compartment_names`. As mentioned in the last section, the signal values within these compartments are often combined together according to the logic defined within a node's `.step()` simulation function. Furthermore, each node contains two other data members of particular interest – the `.connected_cables` list and the `.constant_names` list. The `.constant_names` contains fixed integer/scalar coefficients/values that are also used within an node's `.step()` logic, such as biological constants derived from experimental data or user-set coefficients that can be defined before simulation (like an integration time constant). The `.connected_cables` is an (unordered) list of `Cable` objects that connect to a particular node (one can iterate over this list and print out the names of each cable if need be). Each cable object, which we will discuss in more detail later, has knowledge of the specific compartment within a given node it is to deposit its information into and the node can easily query the name of this compartment by accessing the cable's data member `.dest_comp`.

Given the information above (with the aid of a few other internal book-keeping data structures), a node, after its own `.compile()` routine has been executed (which is done within an `NGCGraph`'s `.compile()` function call), will run its own internal logic each time its `.step()` is called, continually integrating information from its named compartments until the end of simulation time window. While we will defer the exact details of how a `.step()` function is/should be implemented for a subsequent tutorial lesson (which will aid developers interested in contributing their own node types to ngc-learn), we can briefly speak to the neural dynamics that occurs within `.step()` for the two nodes you will work with in this lesson.

For a state node (`SNode`), as seen in its *API*, we see that we have six compartments, which can be printed to I/O as in the following code snippet/example (you can place it in a script named `test_node.py`):

```python
import tensorflow as tf
import numpy as np
from ngclearn.engine.nodes.snode import SNode

a = SNode(name="a", dim=1, beta=1, leak=0.0, act_fx="identity")
print("Compartments:  {}".format(a.compartment_names))
```

which will print the compartments internal to the above node `a`:

```
Compartments:  ['dz_bu', 'dz_td', 'z', 'phi(z)', 'S(z)']
```

We will discuss the first four since the last one is a specialized compartments only used in certain situations. The neural dynamics of a state node, according to the first four compartments, is mathematically depicted by the following partial differential equation:

$$\frac{\partial \mathbf{z}}{\partial t} = -\gamma_{leak}\mathbf{z} + (\mathbf{dz}_{td} + \mathbf{dz}_{bu} \odot \phi'(\mathbf{z})) + \text{prior}(\mathbf{z})$$

where we also formally represent the compartments `dz_bu`, `dz_td`, `z`, and `phi(z)` as $\mathbf{dz}_{bu}$, $\mathbf{dz}_{td}$, $\mathbf{z}$, and $\phi(\mathbf{z})$, respectively. This means that, if we use Euler integration to update the `SNode`'s compartment $\mathbf{z}$ (the default in ngc-learn), $\mathbf{z}$ is updated each call to `.step()` as follows:

$$\mathbf{z} \leftarrow \zeta\mathbf{z} + \beta\frac{\partial \mathbf{z}}{\partial t}$$
$$\phi(\mathbf{z}) = tanh(\mathbf{z}) \quad \text{// } \phi(\mathbf{z}) \text{ can be any activation function}$$

and finally, after $\mathbf{z}$ is updated, the state node will apply an element-wise nonlinear function to $\mathbf{z}$ to get $\phi(\mathbf{z})$ (which is also the name of the fourth compartment). Note that, in the above, we see several of the node's key constants defined, i.e. β or `.beta` (the strength of perturbation applied to the node's $\mathbf{z}$ compartment), γ_{leak} or `.leak` (the strength of the amount of decay applied to the $\mathbf{z}$ compartment's value), and ζ or `.zeta` (the amount of recurrent carry-over or how "stateful" the node is – if one sets the constant `.zeta = 0`, the node becomes "stateless"). prior($\mathbf{z}$) just refers to a distribution function that can be applied to the $\mathbf{z}$ compartment (see *Walkthrough #4* for how this is used/set). We see by observing the above differential equation that a state node is primarily defined by the value of its $\mathbf{z}$ compartment and how this compartment evolves over time is dictated by several factors including the other two compartments $\mathbf{dz}_{td}$ and $\mathbf{dz}_{bu}$ ($\phi'(\mathbf{z})$ refers to the first derivative of the `SNode`'s activation function $\phi(\mathbf{z})$ which can be turned off if desired). Note that multiple cables can feed into $\mathbf{dz}_{td}$ and $\mathbf{dz}_{bu}$ (multiple deposits would be summed to create a final value for either compartment).

As we can see in the above dynamics equations, a state node is simply a set of rate-coded neurons that update their activity values according to a linear combination of several "pressures", notably the two key pressures $\mathbf{dz}_{td}$ (dz_td) and $\mathbf{dz}_{bu}$ (dz_bu) which are practically identical except that dz_bu is a pressure (optionally) weighted by the state node's activation function derivative $\phi'(\mathbf{z})$. In a state node, when you wire other nodes to it, the .step() function will specifically assume that signals are only ever being deposited into either dz_td or dz_bu and NOT into $\mathbf{z}$ (or z) and $\phi(\mathbf{z})$ (or phi(z)), since these last two compartments being evolved according to the equations presented earlier – note that if you accidentally "wire" another node to the z or phi(z) compartments, the SNode will simply ignore those since its .step() function only assumes dz_td and dz_bu receive signals externally).

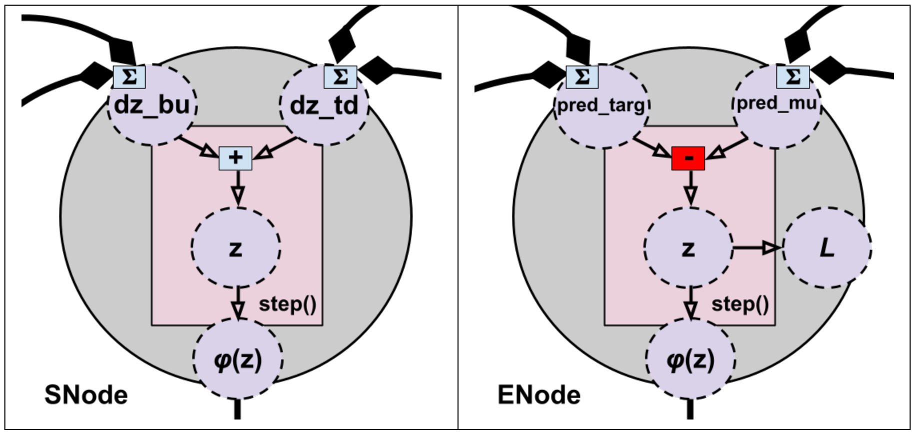
With the SNode above, you can already build a fully functional NGC system (for example, a Harmonium as in *Walk-through #6*), however, there is one special node that we should also describe that will allow you to more easily construct arbitrary predictive coding systems. This node is known as the error node (ENode) and, as seen in its *API*, it contains the following key compartments – pred_mu, pred_targ, z, phi(z), and L or, formally, $\mathbf{z}_{\mu}$, $\mathbf{z}_{targ}$, $\mathbf{z}$, $\phi(\mathbf{z})$, and $L(\mathbf{z})$. An error node is, in some sense, a convenience node because it is actually mathematically a simplification of a state node that is evolved over a period of time (it is a derived "fixed-point" of a pool of neurons that compute mismatch signals evolved over several simulation time steps) and is particularly useful when we want to simulate predictive coding systems faster (and when one is not concerned with the exact biological implementation of neurons that compute mismatch signals but only with their emergent behavior).

The error node dynamics are considerably simpler than that of a state node (and, since they are driven by a derived fixed-point calculation, they are stateless) and simply dictated by the following:

$$\begin{aligned}
\mathbf{z} &= \mathbf{z}_{\mu} - \mathbf{z}_{targ} \\
\phi(\mathbf{z}) &= identity(\mathbf{z}) \quad \text{// } \phi(\mathbf{z}) \text{ can be any activation function} \\
L(\mathbf{z}) &= \sum_j (\mathbf{z} \odot \mathbf{z})^2_j \text{// Gaussian error neurons}
\end{aligned}$$

where $\odot$ denotes elementwise multiplication and $\mathbf{z}_{targ}$ (or pred_targ) is the target signal (which can be accumulated from multiple sources, i.e., if more than cable feeds into it, the set of deposits are summed to create the final compartment value of pred_targ) and $\mathbf{z}_{\mu}$ or (pred_mu) is the expectation of the target signal (which can also be the sum of multiple deposits from multiple cables/sources, i.e., multiple deposits from multiple cables will be summed to calculate the final value of pred_mu). Note that for $L(\mathbf{z})$ (or L), we only depict one possible form that this compartment can take – the Gaussian error neuron (which results in a local mean squared error loss) – although are forms are possible (such as the Laplacian error neuron).

Below, we graphically depict the SNode (Left) and the ENode (Right):



notice that both diagrams indicate that multiple incoming signals (each indicated by a curved diamond-head arrow) are summed within the cell body compartment they are deposited into with the Σ symbol. In the SNode, the signals dz_td and dz_bu are combined by addition, i.e., + (in the light blue box), whereas in the ENode, the signals pred_targ and pred_mu are combined by subtraction, i.e., − (in the red box) (they are contrasted to produce a mismatch/difference signal).

While we do not touch on it in this tutorial lesson, a user could write their own custom nodes as well, making sure to subclass the Node class and then define the dendritic calculation that they require within .step() and ensuring that their custom node writes to the Node class's core compartment data structures so that ngc-learn can effectively simulate the node's evolution over time. Writing one's own custom node will be the subject of an upcoming ngc-learn tutorial lesson.

3.2.2 The Cable

Given the above understanding of a node, all that remains is to combine pairs of them together with an object known as the *cable*. Note that all cables fundamentally are responsible for one particular job: taking the information in one compartment of one "source node", doing something to this information (such as transforming with a bundle of synapses via linear algebra operations), and then depositing this information into the compartment of another "destination node". To do this, there are two primary types of cables you should be familiar with: 1) the simple cable *SCable*, and 2) the dense cable *DCable*. The simple cable simply transmits information directly from one node's compartment to another node's compartment, simply multiplying the information from the source node by its scalar data member .coeff (by default this is set to the value of 1). The dense cable, in contrast, is a bit more involved as it takes the information in one node's compartment and applies some variant of a linear transformation to this signal before depositing it into the compartment of another node (if you wanted a cable to do something more complex than this, you could, as you can for the Node class, write your own custom cable, but we leave this as the subject for a future upcoming tutorial lesson).

Building cables is primarily done with the wire_to() function of the Node class – using this function also makes the destination node aware of the cable that connects to it. Let us say we have two state nodes a and b and we wanted to wire them together such that the information in the z compartment of a is transformed along a dense cable and finally deposited into the dz_td compartment of state node b. This could be done with the following code snippet (place the code in a script named test_cable.py):

```
import tensorflow as tf
import numpy as np
from ngclearn.engine.nodes.snode import SNode

# create the initialization scheme (kernel) of the dense cable
init_kernels = {"A_init" : ("gaussian",0.025)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}

# note that the dim of a does NOT have to equal that of b if using a dense cable
a = SNode(name="a", dim=5, beta=1, leak=0.0, act_fx="identity")
b = SNode(name="b", dim=5, beta=1, leak=0.0, act_fx="identity")
a_b = a.wire_to(b, src_comp="z", dest_comp="dz_td", cable_kernel=dcable_cfg) # wire a to␣
↪b

print("Cable {} of type *{}* transmits from:".format(a_b.name, a_b.cable_type))
print("Node {}.{}".format(a_b.src_node.name, a_b.src_comp))
print(" to ")
print("Node {}.{}".format(a_b.dest_node.name, a_b.dest_comp))
```

which would print to your terminal the following:

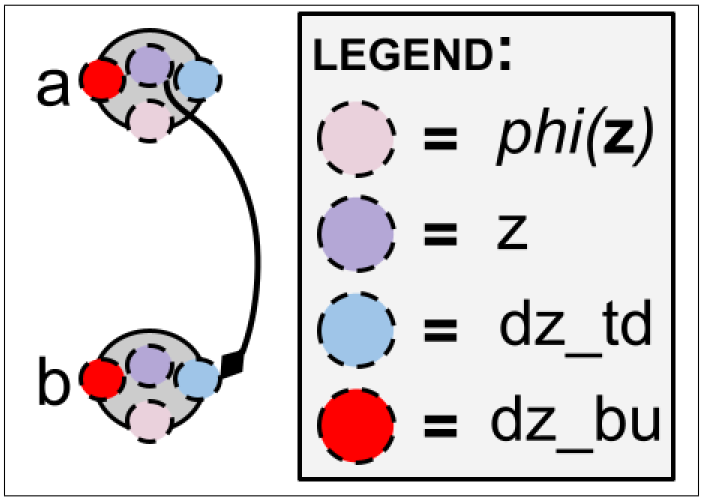
```
Cable a-to-b_dense of type *dense* transmits from:
Node a.z
```

(continues on next page)

(continued from previous page)

```
 to
Node b.dz_td
```

Graphically, the above 2-node circut would look like what is depicted in the figure below.



Note that cables can auto-generate their own `.name` based on the source and destination node that they wire to (in the case above, the cable `a_b` would auto-generate the name `a-to-b_dense`). If you want the cable that wires `a` to `b` to be named something specific, you set the extra argument `name` in `wire_to()` to the desired string and force that cable to take on the name you wish (make sure you choose a unique name). Furthermore, note that a `DCable` has two learnable synaptic objects you can trigger depending on how you initialize the cable:

1) a matrix `A` representing the bundle of synaptic connections that will be used to transform the source node of the cable and relay this information to the destination node of the cable, and
2) a bias vector `b` representing the shift added to the transformed output signal of the cable.

What, then, does the above `a_b` dense cable do mathematically? Let us label `z` compartment of node `a` as $\mathbf{z}^a$ and the `dz_td` of node `b` as $\mathbf{dz}^b_{td}$. Given this labeling, the dense cable will perform the following transformation:

$$\mathbf{s}_{out} = \mathbf{z}^a \cdot \mathbf{A}^{a\_b}$$
$$\mathbf{dz}^b_{td} = \mathbf{dz}^b_{td} + \mathbf{s}_{out}$$

where $\cdot$ denotes a matrix/vector multiplication and $\mathbf{A}^{a\_b}$ is the matrix containing the synapses connecting the compartment `z` of node `a` to the `dz_td` compartment of node `b`. If we had initalized the `DCable` earlier to have a bias, like so:

```
init_kernels = {"A_init" : ("gaussian",0.025), "b_init" : ("zeros")}
```

then the cable `a_b` would perform the following:

$$\mathbf{s}_{out} = \mathbf{z}^a \cdot \mathbf{A}^{a\_b} + \mathbf{b}^{a\_b}$$
$$\mathbf{dz}^b_{td} = \mathbf{dz}^b_{td} + \mathbf{s}_{out}$$

Notice that the last line in the above two equations also shows what each cable will ultimately to node `b` – they add in their transformed signal $\mathbf{s}_{out}$ to its $\mathbf{dz}^b_{td}$ compartment.

If you want to verify that the cable you wired from `a` to `b` appears within node `b`'s `.connected_cables` data member, you can add/write a print statement as follows:

```
print("Cables that connect to Node {}:".format(b.name))
for cable in b.connected_cables:
    print(" => Cable:  {}".format(cable.name))
```

which would print to the terminal:

```
Cables that connect to Node b:
 => Cable:  a-to-b_dense
```

Note that nodes a and b do not have to have the same .dim values if you are wiring them together with a dense cable. In addition, cables in ngc-learn are directional – if you wire node a to node b, this does NOT mean that node b is wired to node a (you would have to call the wire_to() funciton again and create such a wire if this relationship is desired).

If you wanted to wire information directly from node a to node b WITHOUT transforming the information via synapses, you can use a simple cable but, in order to do so, the .dim data member (the number of neurons) of a must be equal to that of b. You could write the following code (in a script you name test_cable2.py):

```
import tensorflow as tf
import numpy as np
from ngclearn.engine.nodes.snode import SNode

# create the initialization scheme (kernel) of the simple cable
scable_cfg = {"type": "simple", "coeff": 1.0}

## Note that you could do the exact same thing with a dense cable using
##   the two lines below but you would be wasting a matrix multiplication if so
# init_kernels = {"A_init" : ("diagonal",1)}
# dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}

# note that the dim of a MUST be equal to b if using a simple cable
a = SNode(name="a", dim=5, beta=1, leak=0.0, act_fx="identity")
b = SNode(name="b", dim=5, beta=1, leak=0.0, act_fx="identity")
a_b = a.wire_to(b, src_comp="z", dest_comp="dz_td", cable_kernel=scable_cfg) # wire a to␣
↪b

print("Cable {} of type *{}* transmits from:".format(a_b.name, a_b.cable_type))
print("Node {}.{}".format(a_b.src_node.name, a_b.src_comp))
print(" to ")
print("Node {}.{}".format(a_b.dest_node.name, a_b.dest_comp))
```

which would print to your terminal the following:

```
Cable a-to-b_simple of type *simple* transmits from:
Node a.z
 to
Node b.dz_td
```

Wiring nodes with cables using the .wire_to() routine notably returns the cable that it creates (in our code snippet this was stored in the variable a_b). This is particularly useful if you need/want to set other properties of the generated cable object such as local Hebbian synaptic update rules, constraints to be applied to the cable's synapses, or synaptic value decay.

3.3 Building Circuits with Nodes with Cables

Once you have created a set of nodes and wired them together in some meaningful fashion, your circuit is now ready to be simulated. To make the circuit function as a complete NGC dynamical system, you must place your nodes into ngc-learn's simulation object, i.e., the `NGCGraph`. This object will, once you have initialized it and made it aware of the nodes you want to simulate, run some basic checks for coherence, internally configure the computations that will drive the simulation that can leverage Tensorflow 2 static graph optimization (you can turn this off if you do not want this optimization to happen), and trigger the compilation routines inherent to each node and cable.

Specifically, if you wanted to compile the simple circuit you created in the last section into a simulated NGC graph, you would then need to write the following (you could add the following lines of code to your `test_cable.py` or `test_cable2.py` scripts if you like to test the compile routine):

```
from ngclearn.engine.ngc_graph import NGCGraph

circuit = NGCGraph()
circuit.set_cycle(nodes=[a,b]) # make the graph aware of nodes a and b, in that order
circuit.compile(batch_size=1)
```

where we see that the graph `circuit` is made aware of nodes `a` and `b` through the call to `.set_cycle()` which takes in as argument a list of `Node` objects. Notice that we do not have to explicitly tell the `NGCGraph` about the cable `a_b` we created – the `NGCGraph` will automatically handle the cable `a_b` through the `.connected_cables` data member of all nodes it is made aware. The `.compile()` routine will desirably do most of the heavy-lifting without much input from the user except for a few small arguments if desired. For example, in the code snippet above, we set the `batch_size` argument directly (the default for an `NGCGraph` if you do not set it is also `1`), which is needed for the default static graph optimization that the `NGCGraph` will set up after you call `.compile()` – note this also means you must make sure that the (mini-)batch size of all sensory inputs you provide to the `NGCGraph` are the of length `batch_size` (since ngc-learn makes use of in-place memory operators to speed up simulation and play nicely with Tensorflow's static graph functionality).

If you do not wish to use the default static graph optimization and be able to deal with variable-length mini-batches of data, then you can replace the above call to `.compile()` by setting its `use_graph_optim` argument to `False` (which has the trade-off that your simulations being slower).

Note that you can always "re-compile" an `NGCGraph` anytime you want. For example, you wish to use the static graph optimization to speed up the training of your `NGCGraph` circuit (since that is the most expensive part of simulating a stateful neural system) but would like to reuse the trained graph on some new pool of data samples with a different mini-batch size (or even, say, online, where you feed in samples to the circuit one at a time). You would simply write the code snippet exactly as we did earlier, run your simulation of the training process, and then, after your code decides that training is done, you could then simply re-compile your simulation object to be dynamic (switching to Tensorflow eager execution mode) as follows:

```
circuit.compile(use_graph_optim=False) # re-compile "circuit" to work w/ dynamic batch␣
↪sizes
```

and you can then present inputs to your simulation object of any batch size you wish. Alternatively, if you still wanted the benefit of the speed offered by static graph optimization but just want to change the batch size to something different than what was used during training (say you have a test set you want to sample mini-batches of `128` samples instead), then you would write the following line:

```
## NOTE: you can also re-compile your circuit to become a system with the same synaptic
## parameters but static-graph optimized for a different fixed batch size (w/o speed␣
↪loss)
circuit.compile(batch_size=128) # <-- note all future batches of data must be length 128
```

re-compiling (as in the above two cases) provides some flexibility to the experimenter/developer although a small setup cost is paid each the `.compile()` routine is called.

Also, it is important to be aware that the `NGCGraph` itself internally maintains several data structures that help it keep track of the simulated nodes/cables, allow it to compute any desired synaptic updates, and ensure that the internal dynamics interact properly with Tensorflow's static graph optimization while still providing inspectability for the experimenter among other activities. One particular object that will be of interest to you, the experimenter, is the `.theta` list, which is the implementation of the mathematical construct Θ often stated in statistical learning and applied mathematics that houses ALL of the learnable parameters (currently it would be empty in our case above because we have not set any learning rules as we will later).

Given the above `NGCGraph`, you have now built your first, very own custom NGC system. All that remains is to learn how to use an NGC system to process some data, which we will demonstrate in the next section.

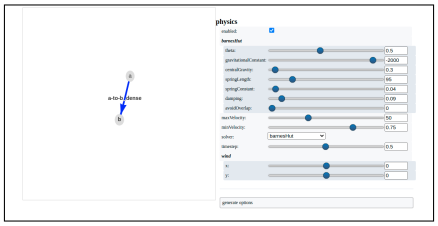
3.3.1 Generating an NGCGraph Visualization

Currently, ngc-learn offers some basic support for generating a visualization of the system architecture that you create with the nodes-and-cables system. This functionality is built on top of the two Python packages `networkx` and `pyviz` to provide the user/experimenter some interactive flexibility with modifying the generated architecture/graph visualizations before saving to disk.

To generate a graphical visualization of your `NGCGraph`, such as one for the 2-node circuit you built in the last section, you would write the following code:

```python
import ngclearn.utils.experimental.viz_graph_utils as viz

viz.visualize_graph(circuit) # generate the graph visual of
```

which will generate a graph/network visualization (after some minor manipulation from the user) similar to the one below:



physics
enabled: ✓
barnesHut
theta: 0.5
gravitationalConstant: -2000
centralGravity: 0.3
springLength: 95
springConstant: 0.04
damping: 0.09
avoidOverlap: 0
maxVelocity: 50
minVelocity: 0.75
solver: barnesHut
timestep: 0.5
wind
x: 0
y: 0
generate options

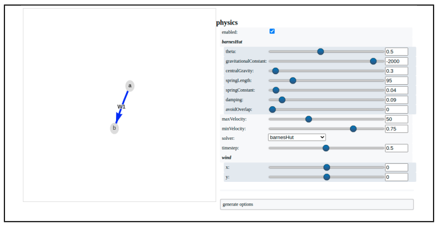
Notice that the node names we set earlier, e.g., `a` and `b`, are automatically extracted by the graph visualizer and the cable names (normally auto-generated) by the `NGCGraph` graph object are attached to the edges they correspond to. Furthermore, observe that you can directly interact with and manipulate (through clicking and dragging) the generated visualization to suit your purposes. Note: we recommend experimenting with the physics `solver` option to the `forceAtlas2Based` or `repulsion` variants for more complex NGC network graphs.

The visualization scheme according to ngc-learn dictates that non-learnable cables are colored blue, dense cables are solid arcs, and that state nodes are colored as as grey ellipses. See the end of this tutorial lesson for more details on the graph color-coding scheme used by ngc-learn.

One important trick to cleaning up an `NGCGraph`'s visualization is to use the `short_name` optional argument to the `.wire_to()` function. Specifically, setting a `short_name` for a particular cable that wires together two nodes allows you to assign "nicknames" to cables while preserving their original auto-generated names (though you can also directly set the names yourself using the `name` argument in the `.wire_to()` routine if you like, just ensure your name choices are unique). For example, we could have created the cable `a_b` earlier with a `short_name` like so:

```
a_b = a.wire_to(b, src_comp="z", dest_comp="dz_td", cable_kernel=scable_cfg, short_name=
↪"W1")
```

If you run the visualizer now but with the `short_name` we set above, you will get the following output:



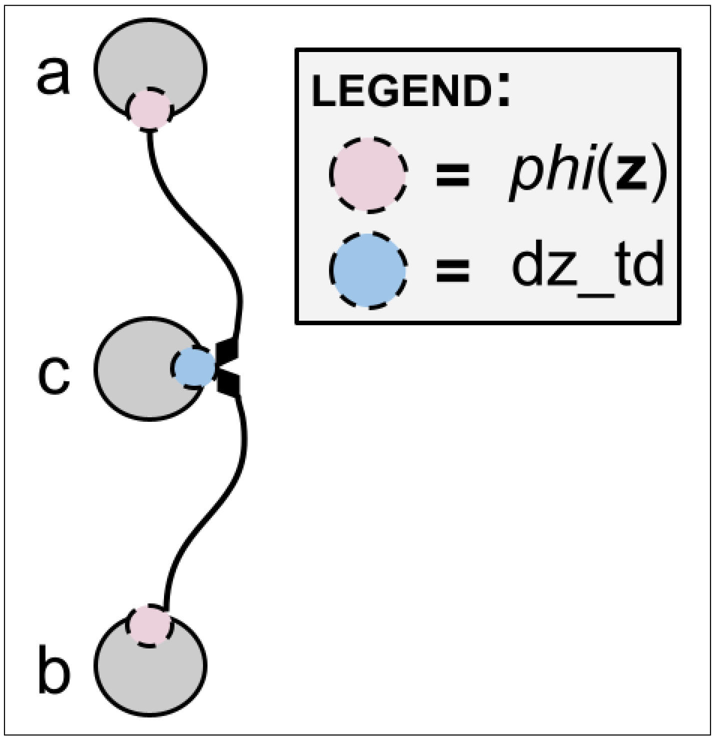
physics
enabled: ✓
barnesHut
theta: 0.5
gravitationalConstant: -2000
centralGravity: 0.3
springLength: 95
springConstant: 0.04
damping: 0.09
avoidOverlap: 0
maxVelocity: 50
minVelocity: 0.75
solver: barnesHut
timestep: 0.5
wind
x: 0
y: 0

generate options

where now `W1` is used in place of the original `a-to-b_dense` auto-generated name.

3.4 Simulating an NGC Circuit with Sensory Data

In this section, we will illustrate two ways in which one may have an `NGCGraph` interact with sensory data patterns. Let us start by building a simple 3-node circuit, i.e., the one depicted in the figure below (only the relevant compartments in each node that we will wire together are depicted).



Create a Python file/script named `circuit1.py` and write the following to create the header:

```python
import tensorflow as tf
import numpy as np

# import building blocks
```

(continues on next page)

(continued from previous page)

```
from ngclearn.engine.nodes.snode import SNode
# import simulation object
from ngclearn.engine.ngc_graph import NGCGraph
```

Now write the following code for your circuit:

```
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
a = SNode(name="a", dim=1, beta=1, leak=0.0, act_fx="identity",
          integrate_kernel=integrate_cfg)
b = SNode(name="b", dim=1, beta=1, leak=0.0, act_fx="identity",
          integrate_kernel=integrate_cfg)
c = SNode(name="c", dim=1, beta=1, leak=0.0, act_fx="identity",
        integrate_kernel=integrate_cfg)

init_kernels = {"A_init" : ("diagonal",1)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
c_b = c.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)

circuit = NGCGraph(K=5)
# execute nodes in order: a, c, then b
circuit.set_cycle(nodes=[a,c,b])
circuit.compile(batch_size=1)

# do something with the circuit above
a_val = tf.ones([1, circuit.getNode("a").dim]) # create sensory data point *a_val*
c_val = tf.ones([1, circuit.getNode("c").dim]) # create sensory data point *c_val*
readouts, _ = circuit.settle(
                clamped_vars=[("a","z",a_val), ("c","z",c_val)],
                readout_vars=[("b","phi(z)")]
              )
b_val = readouts[0][2]
print(" => Value of b.phi(z) = {}".format(b_val.numpy()))
print("             Expected = [[10]]")
circuit.clear()
```

The above (fixed) circuit will simply take in the current values within the `phi(z)` compartment of nodes `a` and `c` and combine them together (through addition) within the `dz_td` compartment of node `b`. Specifically, the value within the `phi(z)` compartment of `a` will be transformed with the dense cable `a_b` and deposited first into `dz_td` of `b` and then the compartment `phi(z)` of `c` will be transformed by the dense cable `c_b` and added to the current value of and deposited into the `dz_td` compartment of `b`. Notice that we set the graph to execute the nodes in a particular order: `a, c, b` so that way we ensure that first the values within nodes `a` and `c` are first computed at any time step followed by node `b` which will then take the current compartment values it needs from `a` and `c` and aggregate them to compute its new state. Alternatively, you could write and set up the same exact computation by organizing the node computation into two subsequent cycles as follows:

```
circuit = NGCGraph(K=5)
# execute nodes in order: a, c, then b
circuit.set_cycle(nodes=[a,c])
circuit.set_cycle(nodes=[b])
circuit.compile(batch_size=1)
```

where the above code-snippet is more explicit and, internally within the `NGCGraph` simulation object, means that a

separate computation cycle will be created that must
wait on the first cycle (a then b) to be completed before it can then be executed (note that the overall simulation needed for both would be the same when finally run).

Now go ahead and run your `circuit1.py` (i.e., `$ python circuit1.py`) and you should get the exact following output in your terminal:

```
=> Value of b.phi(z) = [[10.]]
            Expected = [[10.]]
```

The above output should make sense since we clamped to the `phi(z)` compartments of nodes `a` and `c` vectors of ones, after we run the `NGCGraph` for `K = 5` steps of simulation time within the call to `.settle()`, we should obtain a vector with `10` inside of it for the `phi(z)` compartment of node `b`. This is because, at each time step within the `.settle()` function, the `dz_td` compartment of node `b` is computed according to the following equation:

$$\begin{aligned}\frac{\partial \mathbf{z}^b}{\partial t} &= \phi(\mathbf{z}^a) \cdot \mathbf{A}^{a\_b} + \phi(\mathbf{z}^c) \cdot \mathbf{A}^{c\_b} \\ &= 1 \cdot \mathbf{A}^{a\_b} + 1 \cdot \mathbf{A}^{c\_b} = 1 \cdot \mathbf{I} + 1 \cdot \mathbf{I} \\ &= (1 \cdot 1) + (1 \cdot 1) = 2\end{aligned}$$

where $\mathbf{I}$ is the identity matrix (or diagonal matrix) of size `(1,1)` which is the same as the scalar `1` (because we set the initialize of the `A` matrix within cables `a_b` and `c_b` to be the diagonal matrix). This means that at any time step, nodes `a` and `b` are combined ultimately depositing a scalar value of `2` into node `b`'s `dz_td` compartment, which will then be added according to `b`'s state dynamics: $\mathbf{z}^b \leftarrow \mathbf{z}^b + \beta(\mathbf{dz}^b_{bu} + \mathbf{dz}^b_{td}) = \mathbf{z}^b + \beta(0 + \mathbf{dz}^b_{td})$.

If this calculation is repeated five times, as we have set the `NGCGraph` to do via the argument `K=5`, then the circuit above is effectively repeatedly adding `2` to the `z` compartment of node `b` five times (`2 * 5 = 10`). Note that for node `b`, `phi(z)` is identical to the value of `z` because we set the activation function of node `b` to be $\phi(\mathbf{z}) = \mathbf{z}$ or `act_fx = identity` (in fact, we have done this for all three nodes in this example).

Now, let us slightly modify the above 3-node circuit code to go one step below the application programming interface (API) of the `.settle()` and write our own explicit step-by-step simulation so that way we can examine the value of the `z` and `phi(z)` compartments of node `b` to prove that we are indeed accumulating a value of `2` each time step. To write a low-level custom simulation loop that does the same thing as the code snippet we wrote earlier, you could replace the call to `.settle()` with the following code instead:

```python
# ... same initialization code as before ...

# do something with the circuit above
a_val = tf.ones([1, circuit.getNode("a").dim])
c_val = tf.ones([1, circuit.getNode("c").dim])

circuit.clamp([("a","z",a_val), ("c","z",c_val)])
circuit.set_to_resting_state()
for k in range(K):
    values, _ = circuit.step(calc_delta=False)
    circuit.parse_node_values(values)
    b_val = circuit.extract("b","z")
    print(" t({}) => Value of b.phi(z) = {}".format(k, b_val.numpy()))
print("                  Expected = [[10.]]")
circuit.clear()
```

which will now print out to the terminal:

```
t(0) => Value of b.phi(z) = [[2.]]
t(1) => Value of b.phi(z) = [[4.]]
t(2) => Value of b.phi(z) = [[6.]]
t(3) => Value of b.phi(z) = [[8.]]
t(4) => Value of b.phi(z) = [[10.]]
                 Expected = [[10.]]
```

showing us that, indeed, this circuit is incrementing the current value of the `z` compartment by `2` each time step. The advantage to the above form of simulating the stimulus window for the 3-node instead of using `.settle()` is that one can now explicitly simulate the NGC system online if needed. This lower-level way of simulating an NGC system would be desirable for very long simulation windows where events might happen that interrupt or alter the settling process.

One final item to notice is that, in all of the code-snippets of this section, after the `NGCGraph` has been simulated (either through `.settle()` or online via `.step()`), we call the simulation objects `.clear()` routine. This is absolutely critical to do after you simulate your NGC system for a fixed window of time *IF* you do not want the current values of its internal nodes to carry over to the next time that you simulate the system with `.settle()` or `.step()`. Since an NGC system is stateful, if you expect its internal neural activities to have gone back to their resting states (typically zero vectors) before processing a new pattern or batch of data, then you must make sure that you call `.clear()`. A typical design pattern for an NGC system would something like:

```
# ... initialize the circuit and your optimizer *opt* earlier ...

# after sampling some data, a typical process loop would be:
readouts, delta = circuit.settle( ... ) # conduct iterative inference
opt.apply_gradients(zip(delta, circuit.theta)) # update synapses
circuit.clear() # set all nodes in system back to their resting states
```

3.5 Evolving a Circuit over Time

3.5.1 Shared/Linked Cables

While cables are intended to be unique in that they instantiate a particular bundle of synapses that relay the information from one node to another, it is sometimes desirable to allow two or more cables to reuse the exact same synapse (pointing to the same spot in memory – in other words, they make use of a shallow copy of the synapses). This can also be useful if one needs to reduce the memory footprint of their NGC system, e.g., for CPUs/GPUs with limited memory. To facilitate sharing, you will need to use the `mirror_path_kernel` argument of the `wire_to()` function you used earlier (in place of the `cable_kernel` argument). This argument takes in a 2-tuple where the first argument is the literal cable object you want to share parameters with and the second argument is a string code/flag that tells ngc-learn which parameters (and how) to share.

In ngc-learn, one can make two cables "share" a bundle of synapses, and even bias parameters, as follows (create a file called `circuit2.py` to place the following code into):

```
import tensorflow as tf
import numpy as np

# import building blocks
from ngclearn.engine.nodes.snode import SNode
# import simulation object
from ngclearn.engine.ngc_graph import NGCGraph

```

(continues on next page)

(continued from previous page)

```
# create some nodes
a = SNode(name="a", dim=1, beta=1, leak=0.0, act_fx="identity")
b = SNode(name="b", dim=1, beta=1, leak=0.0, act_fx="identity")
x = SNode(name="x", dim=1, beta=1, leak=0.0, act_fx="identity")
y = SNode(name="y", dim=1, beta=1, leak=0.0, act_fx="identity")

init_kernels = {"A_init" : ("gaussian",0.1)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 111}

a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
# make cable *x_y* reuse the *A* matrix contained in cable *a_b*
x_y = x.wire_to(y, src_comp="phi(z)", dest_comp="dz_td", mirror_path_kernel=(a_b,"A"))

print("Cable {} w/ synapse A = {}".format(a_b.name, a_b.params["A"].numpy()))
print("Cable {} w/ synapse A = {}".format(x_y.name, x_y.params["A"].numpy()))
```

and you should see printed to your terminal:

```
Cable a-to-b_dense w/ synapse A = [[0.1918097]]
Cable x-to-y_dense w/ synapse A = [[0.1918097]]
```

where we see that the cables `a_b` and `x_y` do indeed have the exact same synaptic matrix of size `1 x 1` even though the cables themselves are completely different and even connect completely different nodes (note that you would need to make sure the `.dim` of node `x` is identical to node `a` and that the `.dim` of node `y` is the same as node `b`, otherwise, you will get a shaping error when the cable is later simulated).

There are other ways to share/point to synapses besides the direct way above. For example, the code below will force cable `b_a` to reuse the transpose of the `A` synaptic matrix of cable `a_b`, as indicated by the second code/flag `A^T` input to the `mirror_path_kernel` argument:

```
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
# make cable *b_a* that reuses the transpose of the *A* matrix contained in cable *a_b*
b_a = b.wire_to(a, src_comp="phi(z)", dest_comp="dz_td", mirror_path_kernel=(a_b,"A^T"))
```

Other useful codes for the `mirror_path_kernel` argument include: `A+b` which shares the `A` matrix and bias `b` of the target cable and `-A^T` which shares the negative transpose of matrix `A` of the target cable.

3.5.2 Synaptic Update Rules

A key element of an NGC system is its ability to evolve with time and learn from the data patterns it processes by updating its synaptic weights. To update the synaptic bundles (and/or biases) inside the cables you use to wire together nodes, you will need to also define corresponding learning rules. Currently, ngc-learn assumes that synapses are adjusted through locally-defined multi-factor Hebbian rules.

To configure a cable, particularly a dense cable, to utilize an update rule, you need to specify the following with the `set_update_rule()` routine:

```
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
a_b.set_update_rule(preact=(a,"phi(z)"), postact=(b,"phi(z)"), param=["A"])
```

where we must define at least three arguments:

1) the pre-activation term `preact` which must be a 2-tuple containing the pre-activation node object and a string stating the compartment that we want to extract a vector signal from,

2) the post-activation term `postact` defined exactly the same as the pre-activation term, and
3) a list of strings `param` stating the synaptic parameters we want the update rule to affect. The code-snippet above will tell ngc-learn that when cable `a_b` is updated, we would like to take the (matrix) product of node `a`'s `phi(z)` compartment and node `b`'s `phi(z)` compartment and specifically adjust matrix `A` within the cable.

If cable `a_b` also contained a bias, we would specify the rule as follows:

```
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
a_b.set_update_rule(preact=(a,"phi(z)"), postact=(b,"phi(z)"), param=["A", "b"])
```

and ngc-learn will intelligently realize that synaptic vector `b` of cable `a_b` will be updated using only the post-activation term `postact` (since it is a vector and not a matrix like `A`).

Using the `.set_update_rule()` function on each cable that you would like to evolve or be updated given data is all that you need to do to set up local learning. The `NGCGraph` will automatically become aware of the valid cables linking nodes that are learnable and, internally, call those cables' update rules to compute the correct synaptic adjustments. In particular, whenever you call `.settle()` on an `NGCGraph`, the simulation object will actually compute ALL of the synaptic adjustments at the end of the simulation window and store them into a list `delta` and return them to you.

For example, you want to compute the Hebbian update for the cable `a_b` earlier (that you wrote for `circuit2.py`) given a data point containing the value of one (create a new file and write the code below into `circuit3.py`):

```
import tensorflow as tf
import numpy as np

from ngclearn.engine.nodes.snode import SNode # import building blocks
from ngclearn.engine.ngc_graph import NGCGraph # import simulation object

# create the initialization scheme (kernel) of the dense cable
init_kernels = {"A_init" : ("gaussian",0.1)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 111}

a = SNode(name="a", dim=1, beta=1, leak=0.0, act_fx="identity")
b = SNode(name="b", dim=1, beta=1, leak=0.0, act_fx="identity")
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
a_b.set_update_rule(preact=(a,"phi(z)"), postact=(b,"phi(z)"), param=["A"])

print("Cable {} w/ synapse A = {}".format(a_b.name, a_b.params["A"].numpy()))

circuit = NGCGraph()
# execute nodes in order: a, c, then b
circuit.set_cycle(nodes=[a,b])
circuit.compile(batch_size=1)

opt = tf.keras.optimizers.SGD(0.01)

# do something with the circuit above
a_val = tf.ones([1, circuit.getNode("a").dim]) # create sensory data point *a_val*
readouts, delta = circuit.settle(
                    clamped_vars=[("a","z",a_val)],
                    readout_vars=[("b","phi(z)")]
                  )
opt.apply_gradients(zip(delta, circuit.theta))
circuit.clear()
```

(continues on next page)

(continued from previous page)

```
print("Update to cable {} is: {}".format(a_b.name, delta[0].numpy()))
```

which would print to your terminal:

```
Update to cable a-to-b_dense is: [[-0.9590485]]
```

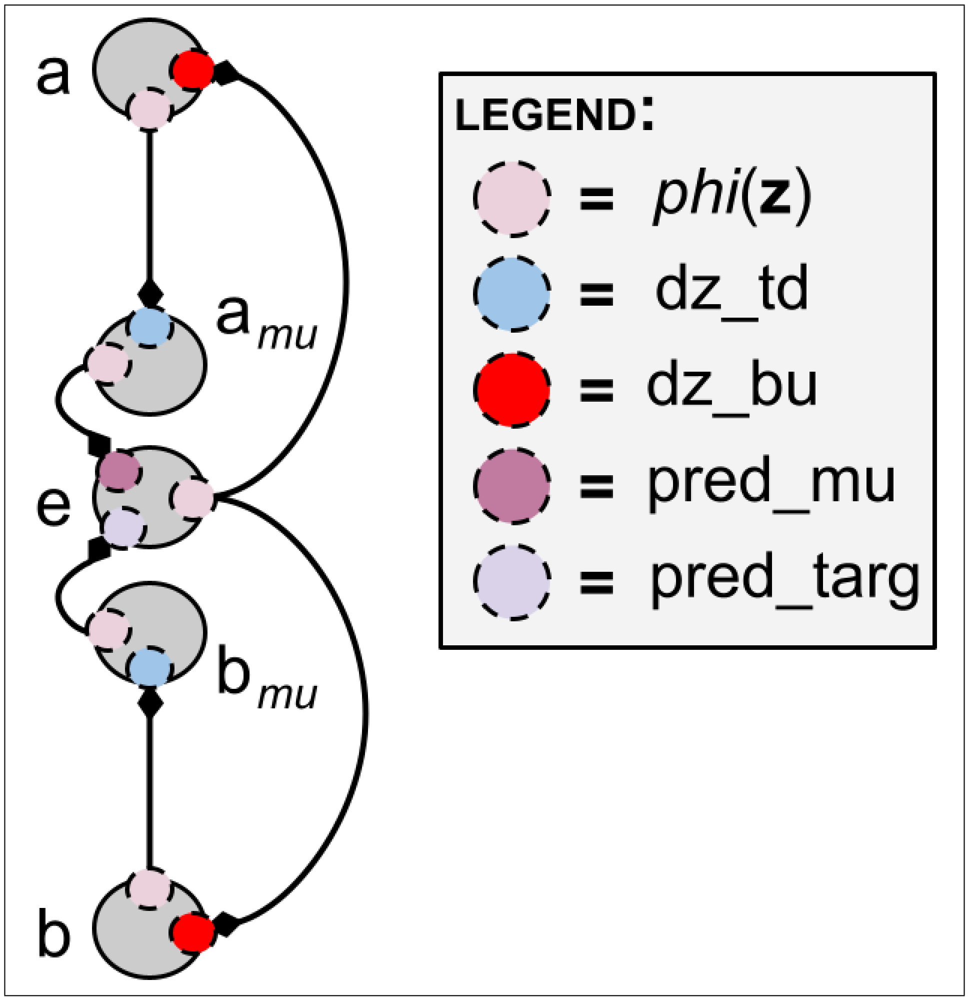
Notice that we have demonstrated how ngc-learn interacts with Tensorflow 2 optimizers by simply giving the returned `delta` list and the circuit's internal `.theta` list to the optimizer which will then physically adjust the values of synaptic bundles themselves for you. NOTE that the order of Hebbian updates will be returned in the exact same order as the learnable parameters that `.theta` points to.

The above NGC system is, of course, rather naive as we would effectively be calculating and update the single synapses that connects nodes `a` and `b`, and, since this use of the update rule is classical Hebbian, the value of the synapse inside of `A` of cable `a_b` would grow indefinitely. In the next section, we will craft a more interesting circuit that uses what you learned about with respect cables and nodes, including the error node `ENode`.

3.6 Constructing a Convergent 5-Node Circuit

As our final exercise for this tutorial, let us build a 5-node circuit that attempts to learn how to converge to a state such that a five-dimensional node `a` and a six-dimensional node `b` each generate three-dimensional output values
that are nearly identical. In other words, we want node `a` to get good at predicting the output of node `b` and node `b` to get good at predicting the output of node `a`. Furthermore, node `b`'s `z` compartment will always be clamped to a vector of ones. To measure the mismatch between these two nodes' predictions, we will introduce the fifth and final node as a three-dimensional error node tasked with computing how far off the two sources nodes are from each other.

We illustrate the 5-node circuit in the figure below. The relevant compartments that we will be wiring together are shown as different-colored circles (and the legend maps the color to the compartment name).



To build this circuit, create a file called `circuit4.py` and write the header:

```
import tensorflow as tf
import numpy as np

# import building blocks
from ngclearn.engine.nodes.enode import ENode
from ngclearn.engine.nodes.snode import SNode
# import simulation object
from ngclearn.engine.ngc_graph import NGCGraph
```

and then go ahead and create the 5-node circuit we described as follows:

```
# create the initialization scheme (kernel) of the dense cable
init_kernels = {"A_init" : ("gaussian",0.1)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 111} # dense cable
scable_cfg = {"type": "simple", "coeff": 1.0} # identity cable

a_dim = 5
e_dim = 3
b_dim = 6
```

(continues on next page)

(continued from previous page)

```
# Define node a
a = SNode(name="a", dim=a_dim, beta=1, leak=0.0, act_fx="identity")
# Define node a_mu
a_mu = SNode(name="a_mu", dim=e_dim, beta=1, zeta=0, leak=0.0, act_fx="identity")
# Define error node e
e = ENode(name="e", dim=e_dim)
# Define node b
b = SNode(name="b", dim=b_dim, beta=1, leak=0.0, act_fx="identity")
# Define node b_mu
b_mu = SNode(name="b_mu", dim=e_dim, beta=1, zeta=0, leak=0.0, act_fx="identity")

# wire a to a_mu
a_amu = a.wire_to(a_mu, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
a_amu.set_update_rule(preact=(a,"phi(z)"), postact=(e,"phi(z)"), param=["A"])
# wire a_mu to e
amu_e = a_mu.wire_to(e, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=scable_cfg)

# wire b to b_mu
b_bmu = b.wire_to(b_mu, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
b_bmu.set_update_rule(preact=(b,"phi(z)"), postact=(e,"phi(z)"), param=["A"])
# wire b_mu to e
bmu_e = b_mu.wire_to(e, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=scable_
↪cfg)

# wire e back to a
e_a = e.wire_to(a, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(a_amu,"A^T
↪"))

# wire e back to b
e_b = e.wire_to(b, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(b_bmu,"A^T
↪"))

circuit = NGCGraph()
# execute nodes in order: a, c, then b
circuit.set_cycle(nodes=[a, a_mu, b, b_mu])
circuit.set_cycle(nodes=[e])
circuit.set_learning_order([b_bmu, a_amu]) # enforces order - b_bmu then a_amu
circuit.compile(batch_size=1)

opt = tf.keras.optimizers.SGD(0.05)
```

and then, given the 5-node graph you crafted and compiled above, you can now write a simple training loop to simulate as in below:

```
n_iter = 60 # number of overall optimization steps to take

b_val = tf.ones([1, circuit.getNode("b").dim]) # create sensory data point *b_val*
print("---- Simulating Circuit Evolution ----")
for t in range(n_iter):
    readouts, delta = circuit.settle(
```

(continues on next page)

(continued from previous page)

```
                        clamped_vars=[("b", "z",b_val)],
                        readout_vars=[("e", "L")]
                      )
    e_val = readouts[0][2]
    if t > 0:
        print("\r{} => Value of e.L = {}".format(t, e_val.numpy()),end="")
    else:
        print("{} => Value of e.L = {}".format(t, e_val.numpy()))
    opt.apply_gradients(zip(delta, circuit.theta))
    circuit.clear()
print()

print("---- Final Results ----")
# get final values
readouts, delta = circuit.settle(
                    clamped_vars=[("b", "z",b_val)],
                    readout_vars=[("e", "pred_mu"),("e", "pred_targ")],
                    calc_delta=False # turn off update computation
                  )
prediction = readouts[0][2].numpy()
target = readouts[1][2].numpy()
print("Prediction: {}".format(prediction))
print("    Target: {}".format(target))
circuit.clear()
```

Once you have written `circuit4.py`, you can execute it from the command line as `$ python circuit4.py` which should print to your terminal something similar to:

```
---- Simulating Circuit Evolution ----
0 => Value of e.L = [[0.03118673]]
59 => Value of e.L = [[3.9547285e-05]]
---- Final Results ----
Prediction: [[-2.2072585  -1.1418786   0.68785524]]
    Target: [[-2.2125444 -1.1444474  0.6895305]]
```

As you can see, the loss represented by the error node `e` (specifically, the value stored in its loss `L` compartment), starts at greater than `0.03` and then decreases over the sixty simulated training iterations to nearly zero (`0.00003954`), and, as we can see in the comparison between the prediction from node `a` against the target produced by node `b`, the values are quite close. This indicates that our small 5-node circuit has converged to an equilibrium point where node `a` and node `b` are capable of matching each other (assuming that `b`'s `z` compartment will always be clamped to a vector of ones). Furthermore, we see that we have crafted a feedback loop via cable `e_a`, which transmits the error information contains inside of node `e` back to the `dz_bu` compartment of node `a`, which, as we recall from the earlier part of this tutorial, is used in node `a`'s state update equation. (Feedback loop `e_b` does something similar to `e_a`, however, since we force the `z` compartment of `b` to always be a specific value, this loop ends up being useful in this example).

With the completion of the above example, you have now gone through the process of crafting your own custom NGC circuit with ngc-learn's nodes-and-cables system. Given this knowledge, you are ready to design and simulate your own predictive processing neural systems based on the NGC computational framework. For examples of how nodes and cables are used to build various classical and modern-day models, check out the *Model Museum* (including the pre-designed agents in the ngc-learn repo `ngclearn/museum/`) and the walk-throughs.

3.7 Knowing the Utility Functions of an NGCGraph

Although you have learned of and how to assemble the key elements in ngc-learn needed to construct NGC circuits, there are a few useful utility functions that are provided once you construct the `NGCGraph` simulation object. In this closing section, we will briefly discuss each of these and briefly illustrate their use (and review key ones that we covered earlier).

3.7.1 Compiling and Re-Compiling Your Simulation Object

As discussed earlier in this tutorial lesson, the `.compile()` is one of the most important functions to call after you have constructed your `NGCGraph` as it will set up the crucial internal bookkeeping and checks to ensure that your simulated NGC system works correctly with static graph optimization and is properly analyzable.

Normally, just calling the `.compile()` function after you initialize the `NGCGraph` constructor is sufficient so long as you either set its `batch_size` argument to the batch size you will be training with (and you must ensure that your data is presented to your graph in batch sizes with that exact same length each time, otherwise the `NGCGraph` will throw a memory error). Note that you can also set the batch size your graph expects in the constructor itself, like so `NGCGraph(K=10, batch_size=128)`.

If you do not wish for ngc-learn to use static graph optimization, you can always turn this off by setting the `use_graph_optim` to `False` in the `.compile()` function, which will allow you to use variable-length batch sizes (and not force you to specify the `batch_size` in the compile routine or in the `NGCGraph` constructor) but this will come at the cost of slower simulation time especially if you will be evolving the synapses over time (only in the case of pure online learning might turning off the static graph optimization be useful). However, you can, as was discussed earlier, always "re-compile" your simulation object if, for example, you will be training with mini-batches of one length and then testing with mini-batches of another length. Re-compiling is simple and not too expensive to do if done sparingly – all you need to do is call `.compile()` again and choose a new `batch_size` to give it as an argument.

One final note about the `.compile()` routine is that it actually returns a dictionary of dictionaries that contains/organizes the core specifications of your `NGCGraph`. You can print this dictionary out if you like and examine that the various nodes and cables state the various key properties you expect them to aid in debugging. Future plans for ngc-learn will be to leverage this simulation properties dictionary to aid in auto-generated visualization to help in creating architecture figures and possibly information-flow diagrams (we would also like to mention here that we welcome community contributions with respect to visualization and system analysis if you are interested in helping with this particular effort).

3.7.2 Clearing the State of Your Simulator

Another core routine that you learned about in this tutorial is the `.clear()` function. This is a critical function to call whenever you want to completely wipe out the state of your `NGCGraph` simulation object. Wiping graph state is something you will likely want to do quite often in your code. For example, a typical design pattern for simulating an NGC system after you sample a batch of training data points is to: 1) first call its `.settle()` function, 2) do something with the readout variables you asked it to return (and maybe extract some other items from your graph), 3) update the system's synaptic weights (as housed in its `.theta` construct) using an external optimization algorithm like stochastic gradient descent, 4) apply/enforce constraints, and 5) clear/wipe the graph state.

There are no arguments to `.clear()` but you should be aware that it does wipe the state of your graph thoroughly – this also means that, after clearing, using a getter function `.extract()` (discussed in the next section) becomes meaningless the internal bookkeeping structures that your graph maintains get set to their default ("empty") states. Note that clearing the graph state is **NOT** the same as setting nodes exactly to their resting state – node resting states are actually set with a call to `.set_to_resting_state()` and this is actually done for you every time you call `.settle()` (unless you tell your graph not to start at a resting state by setting the `cold_start` flag argument to `False`).

Note that a use-case where you might *not* want to use the `.clear()` function is if you are simulating an NGC system over one long, single window of time (for example, a sensory data stream). In this scenario, using `.clear()` would be against the processing task as the neural system should be aware of its previous nodal compartment activities after the last call to `.settle()` (you would want to also set `cold_start` to `False` in this situation). We remark that a better alternative to using `.settle()` for streaming data applications is to, like we did early in this tutorial, work with the lower-level API of your `NGCGraph` and just use its `.step()` routine which *exactly* simulates one discrete step of time of your graph. This would allow you to set up "events" such as when you want `.step()` to return updates to synapses (by setting the `calc_delta` argument to `True` if you do and `False` otherwise) and when you want node compartments to go to their actual resting states with a call to `.set_to_resting_state()`. We caution the user that leveraging the lower-level online functionality of an `NGCGraph` does require some degree of comfort with how ngc-learn operates and care should be taken to check that your system is evolving in the way that you expect (working with the online functionality of an NGC system will be the subject of a future advanced lesson). While it offers flexibility, the `.step()` function also assumes that the experimenter will properly set the other functions that `.settle()` normally takes care of automatically, such as `.set_to_resting_state()`, clamping, and injecting compartment values.

3.7.3 Setting the Order of Synaptic Adjustments

Normally, when you set update rules for cables that you would like to evolve with time, your `NGCGraph` will determine its own order in which the calculated adjustments appear in the `delta` object (returned from `.settle()`) as well as the order in which learnable parameters appear in the `.theta` data member. If you wanted the order of the cables to appear in a certain way in `.theta` (which would affect the order of `delta`), you can use the `.set_learning_order()` function before you call the `.compile()` routine for your `NGCGraph`.

This was actually done earlier in the last section, where you set the order of the cable parameters in `.theta` to be cable `b_bmu` followed by cable `a_amu` as in the code snippet reproduced from earlier:

```
circuit.set_learning_order([b_bmu, a_amu]) # enforces order - b_bmu then a_amu
```

Setting the order of learnable cables directly affects what is returned by functions such as `.settle()` and `.step()` since, internally, the `NGCGraph` will organize itself to ensure that the order of updates in `delta` exactly match the order of learnable parameters stored in `.theta`. (Note: if a cable has a synaptic matrix `A` and bias `b`, then always the order will be that cable's `A` followed by `b` in `.theta`.)

3.7.4 Extracting Signals and Properties: Getter Functions

Two of the most important "getter" functions you will want to be familiar with when dealing with `NGCGraph`'s are `.extract()` and `.getNode()`.

The `.extract()` function is useful when you want to access particular values of your NGC system at a particular instant. For example, let us say that you want to retrieve and inspect the value of the `z` compartment of the node `a` in the 5-node circuit you built in the last section right. You would then utilize the `.extract()` methods as follows:

```
node_value = circuit.extract("a", "z")
print(" -> Inspecting value of Node a.z = {}".format(node_value.numpy()))
```

which would print to your terminal:

```
 -> Inspecting value of Node a.z = [[-0.9275531   2.341278    0.2365013   1.2464949   0.
↪76036114]]
```

NOTE: it is meaningless to call `.extract()` in the following two cases:

1) after you call `.clear()`, as `.clear()` will completely wipe the state of your `NGCGraph`, and

2) not before you have simulated used your `NGCGraph` for any amount of time (if you have never simulated the graph, then your graph has no signals of any meaning since it has never interact with data or an environment). If you call `.extract()` in cases like those above, it will simply return `None`.

The `.getNode()` is useful if you have already compiled your `NGCGraph` simulation object and want to retrieve properties related to a particular node in this graph. For example, let us say that you want to determine the dimensionality of the `e` node in your 5-node circuit of the last section. To do this, you would write the following code:

```
node = circuit.getNode("e")
print(" -> The dimensionality of Node e is {}".format(node.dim))
```

which would print to your terminal:

```
 -> The dimensionality of Node e is 3
```

The `.getNode()` method will return the full `Node` object of the same exact name you input as argument. With this object, you can query and inspect any of its internal data members, such as the `.connected_cables` as we did earlier in this lesson.

3.7.5 Clamping and Injecting Signals: Setter Functions

The two "setter" functions that will you find most useful when working with the `NGCGraph` are `.clamp()` and `.inject()`. Clamping and injecting, which both work very similarly, allow you to force certain compartments in certain nodes of your choosing to take on certain values before you simulate the `NGCGraph` for a certain period of time. While both of these initially place values into compartments, there is a subtle yet important difference in the effect each has on the graph over time. Desirably, both of these functions take in a list of arguments, allowing you clamp or inject many items at one time if needed.

In the event that you want a particular node's compartment to take on a specific set of values and **remain fixed at** these values throughout the duration of a simulation time window, then you want to use `.clamp()`. In our 5-node circuit earlier, we in fact did this in our particular call to `.settle()` (which, internally, actually makes a call to `.clamp()` for you if you provide anything to the `clamped_vars` argument), but you could, alternatively, use the clamping function explicitly if you need to as follows:

```
b_val = tf.ones([1, circuit.getNode("b").dim])
circuit.clamp([("b", "z", b_val)])
readouts, delta = circuit.settle(
                    readout_vars=[("e", "pred_mu"),("e", "pred_targ")],
                    calc_delta=False # turn off update computation
                  )

node_value = circuit.extract("b", "z")
print(" -> Inspecting value of Node b.z = {}".format(node_value.numpy()))
```

which will, through each step of simulation conducted within the `.settle()` force the `z` compartment of node `b` to ALWAYS remain at the value of `b_val` (this vector of ones will persist throughout the simulation time window). The result of this code snippet prints to terminal the following:

```
 -> Inspecting value of Node b.z = [[1. 1. 1. 1. 1. 1.]]
```

This is as we would expect – we literally clamped a vector of six ones to `z` of node `b` and would expect to observe that this is still the case at the end of simulation.

If, in contrast, you only want to **initialize** a particular node's compartment to *start* at a specific value but not necessarily remain at this value, you will want to use `.inject()`. Doing so looks like code below:

```
b_val = tf.ones([1, circuit.getNode("b").dim])
circuit.inject([("b", "z", b_val)])
readouts, delta = circuit.settle(
                    readout_vars=[("e", "pred_mu"),("e", "pred_targ")],
                    calc_delta=False # turn off update computation
                  )

node_value = circuit.extract("b", "z")
print(" -> Inspecting value of Node b.z = {}".format(node_value.numpy()))
```

which looks nearly identical to the clamping code we wrote above. However, the result of this computation is quite different as seen in the terminal output below:

```
 -> Inspecting value of Node b.z = [[8.505673  8.249885  8.257135  7.7380524 8.38973   8.
↪267948 ]]
```

Notice that the values within `z` of node `b` are NOT ones like we saw in our previous clamping example. This is because this compartment only started at the first time step as a vector of ones but, according to the internal dynamics of node `b` which are driven by the originally useless feedback loop/cable `e_b` we created earlier – recall, at the time, that we wrote that this cable would do nothing because we *clamped* `z` in node `b` to a vector of ones. If we had instead *injected* the vector of ones, this compartment in node `b` would indeed have evolved over time.

3.7.6 Enforcing Constraints

One final item that you may find important when simulating the evolution of an `NGCGraph` is the enforcing of constraints through the `.apply_constraints()` routine. For example, you want to ensure that the Euclidean norms of the columns of a particular matrix `A` in one of your system's cables never exceed a certain value (see *Walkthrough #4* for a case that requires this constraint to be true).

To enforce a constraint on a particular cable, all you need to do is first make the desired cable aware of this constraint like so:

```
a_amu = a.wire_to(a_mu, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
constraint_cfg = {"clip_type":"norm_clip","clip_mag":1.0,"clip_axis":1}
a_amu.set_constraint(constraint_cfg)
```

then, whenever you call the `.apply_constraints()` of your `NGCGraph` simulation object, this constraint will internally be enforced/applied to the cable `a_amu`. Typically, this call looks like the following (using our 5-node circuit as an example):

```
readouts, delta = circuit.settle(
                    clamped_vars=[("b", "z",b_val)],
                    readout_vars=[("e", "L")]
                  )
opt.apply_gradients(zip(delta, circuit.theta))
circuit.apply_constraints() # generally apply constraints after an optimizer is used...
circuit.clear()
```

where we see that we call `.apply_constraints()` AFTER the Tensorflow optimizer has been used to actually alter the values of the synapses of the NGC system. If, after the `SGD` update had resulted in the norms of any of the columns in the matrix `A` of cable `a_amu` to exceed the value of `1.0`, then `.apply_constraints()` would further alter this matrix to make sure it no longer violates this constraint.

A Note on Synaptic Decay: Like norm constraints, weight/synapse decay is also treated as a (soft) constraint in an NGCGraph. If you want to apply a small decay to a particular synaptic bundle matrix A in a particular cable, you can easily do so by simply calling the .set_decay() function like so:

```
a_b.set_decay(decay_kernel=("l1",0.00005)) # apply L1 weight decay to *A* in cable *a_b*
```

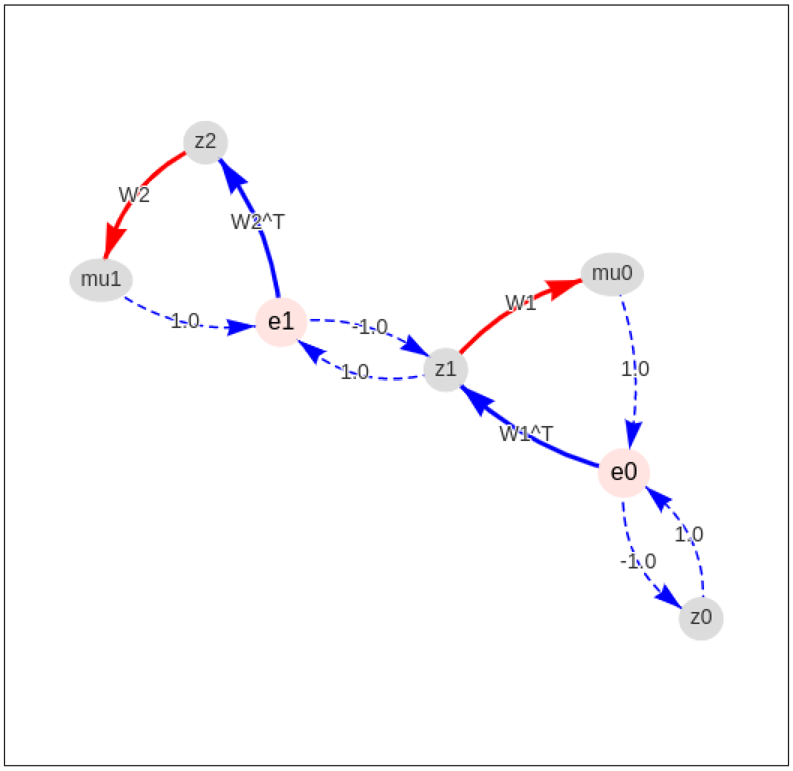
which would apply a decay factor based on a centered Laplacian distribution (or an L1 penalty). If you chose l2 instead, the decay factor applied would then be based on a centered Gaussian distribution (or an L2 penalty) over each element in matrix A of cable a_b.

3.7.7 A Note on Graph Visualization

Earlier, we explored ngc-learn's support for NGC architecture visualization, where we learned about the graph visualizer and using the short_name argument to superimpose desired "nicknames" for particular cables (yielding a less cluttered graph plot). As you build more complex graphs that combine different kinds of nodes, you will see other aspects of ngc-learn's node and cable coloring/visual depiction scheme rendered. In this note, we will briefly define the full scheme:

1) dense cables (DCable) are solid arcs,
2) simple cables (SCable) are dashed arcs,
3) non-learnable/evolving cables are colored blue,
4) learnable/evolving cables are colored red,
5) state nodes are colored gainsboro (or a grayish color),
6) error nodes are colored mistyrose (or light reddish color) with slightly larger text,
7) forward nodes are colored lavender, and
8) spiking nodes are colored antiquewhite.

For example, a visualization of a hierarchical NGC generative model containing both state and error nodes would be the following:



Note that the plotted graph produced by the visualizer is always a directed graph. Furthermore, notice that the `visualize_graph()` method returns a full `networkx` directed graph, amenable to all of the graph operations/network analysis tools available to `networkx` graph objects. You can also alter the output path of the generated dynamic HTML (`*.html`) object by modifying `output_dir`, which will also change the location of a GraphML object saved to disk which is auto-named `<name_of_your_ngcgraph>.graphml` (for use with external graph analysis toolkits that can read in the GraphML file format).

3.8 Conclusion

You now have successfully gone through the core details and functionality of ngc-learn's nodes-and-cables system. The next step is to build your own NGC systems/models for your own research projects and play with the pre-designed systems in the Model Museum (and go through the walkthroughs). In future upcoming tutorial lessons, we will cover topics such designing your own customs nodes or cables that interact with the nodes-and-cables system and working with the low-level online functionality of simulated NGC systems.

3.9 References

Hebb, Donald Olding. The organization of behavior: A neuropsychological theory. Psychology Press, 2005.

CHAPTER

FOUR

WALKTHROUGH 1: LEARNING NGC GENERATIVE MODELS

In this demonstration, we will learn how to use ngc-learn's Model Museum to fit an NGC generative model, which is also called a generative neural coding network (GNCN), to the MNIST database. Specifically, we will focus on training three key models, each from different points in history, and estimate their marginal log likelihoods. Along the way, we will see how to fit a prior to our models and examine how a simple configuration file can be set up to allow for easy recording of experimental settings.

Concretely, after going through this demonstration, you will:

1. Understand how to import and train models from ngc-learn's Model Museum.
2. Fit a density estimator to an NGC model's latent space to create a prior.
3. Estimate the marginal log likelihood of three GNCNs using the prior-fitting scheme designed in this demonstration.

Note that the two folders of interest to this demonstration are:

- `walkthroughs/demo1/`: this contains the necessary scripts and configuration files
- `walkthroughs/data/`: this contains a zipped copy of the MNIST database arrays

4.1 Setting Up and Training a Generative System

To start, navigate to the `walkthroughs/` directory to access the example/demonstration code and further enter the `walkthroughs/data/` sub-folder. Unzip the file `mnist.zip` to create one more sub-folder that contains a set of numpy arrays each house a different slice of the MNIST database, i.e., `trainX.npy` and `trainY.npy` compose the training set (image patterns and their labels), `validX.npy`and `validY.npy` make up the development/validation set, and `testX.npy`and `testY.npy` compose the test set. Note that pixels in all image vectors have been normalized for you, to the range of [0,1].

Next, in `walkthroughs/demo1/`, observe the provided script `sim_train.py`, which contains the code to execute the training process of an NGC model. Inside this file, we can export one of three possible GNCNs from ngc-learn's Model Museum, i.e., the *GNCN-t1* (which is an instantiation of the model proposed in Rao & Ballard, 1999 [1]), the *GNCN-t1-Sigma* (an instantiation of the model proposed in Friston 2008 [2]), and the *GNCN-PDH* (one of the models proposed in Ororbia & Kifer 2022 [3]).

Importing models from the Model Museum is straightforward and only requires a few lines to be placed in the header of a training script. Notice that we import several items besides the models, including a *DataLoader*, like so:

```
from ngclearn.utils.data_utils import DataLoader
```

some metrics, transformations, and other I/O tools, as follows:

```python
from ngclearn.utils.config import Config
import ngclearn.utils.transform_utils as transform
import ngclearn.utils.metric_utils as metric
import ngclearn.utils.io_utils as io_tools
```

where *Config* is an argument configuration object that reads in values set by the user in a `*.cfg` configuration file, `transform_utils` contains mathematical functions to alter vectors/matrices (we will use the `binarize()` function, which, inside of `sim_train.py`, will convert the MNIST image patterns to their binary equivalents), and `metric_utils` contains measurement functions (we will use the binary cross entropy routine `bce()`). Finally, we import the models themselves, as shown below:

```python
from ngclearn.museum.gncn_t1 import GNCN_t1
from ngclearn.museum.gncn_t1_sigma import GNCN_t1_Sigma
from ngclearn.museum.gncn_pdh import GNCN_PDH
```

With the above imported from ngc-learn, we have everything we need to craft a full training cycle as well as track a model's out-of-sample inference ability on validation data.

Notice in the script, at the start of our with-statement (which is used to force the following computations to reside in a particular GPU/CPU), before initializing a chosen model, we define a second special function to track another important quantity special to NGC models – the total discrepancy (ToD) – as follows:

```python
def calc_ToD(agent):
    """Measures the total discrepancy (ToD) of a given NGC model"""
    ToD = 0.0
    L2 = agent.ngc_model.extract(node_name="e2", node_var_name="L")
    L1 = agent.ngc_model.extract(node_name="e1", node_var_name="L")
    L0 = agent.ngc_model.extract(node_name="e0", node_var_name="L")
    ToD = -(L0 + L1 + L2)
    return float(ToD)
```

This function is used to measure the internal disorder, or approximate free energy, within an NGC model based on its error neurons (since, internally, our imported models use the specialized *ENode* to create error neuron nodes, we retrieve each node's specialized compartment known as the scalar local loss `L` – for details on nodes and their compartments, see *Demonstration # 2* for details – but you could also compute each local loss with distance functions, e.g., `L2 = tf.norm( agent.ngc_model.extract(node_name="e2", node_var_name="phi(z)"), ord=2 )`). Measuring ToD allows us to monitor the entire NGC system's optimization process and make sure it is behaving correctly, making progress towards reaching a stable fixed-point.

Next, we write an evaluation function that leverages a `DataLoader` and a NGC model and returns some useful problem-specific measurements. In this demo's case, we want to measure and track binary cross entropy across training iterations/epochs. The evaluation loop can be written like so:

```python
def eval_model(agent, dataset, calc_ToD, verbose=False):
    """
        Evaluates performance of agent on this fixed-point data sample
    """
    ToD = 0.0 # total disrepancy over entire data pool
    Lx = 0.0 # metric/loss over entire data pool
    N = 0.0 # number samples seen so far
    for batch in dataset:
        x_name, x = batch[0]
        N += x.shape[0]
        x_hat = agent.settle(x) # conduct iterative inference
```

(continues on next page)

(continued from previous page)

```
        # update tracked fixed-point losses
        Lx = tf.reduce_sum( metric.bce(x_hat, x) ) + Lx
        ToD = calc_ToD(agent) + ToD # calc ToD
        agent.clear()
        if verbose == True:
            print("\r ToD {0}  Lx {1} over {2} samples...".format((ToD/(N * 1.0)), (Lx/
↪(N * 1.0)), N),end="")
    if verbose == True:
        print()
    Lx = Lx / N
    ToD = ToD / N
    return ToD, Lx
```

Notice that, in the above code snippet, we pass in the current NGC model (agent), the DataLoader (dataset), and the ToD function we wrote earlier. Now we have a means to measure some aspect of the generalization ability of our NGC model, all that remains is to craft a training process loop for NGC model. This loop could take the following form:

```
# create a  training loop
ToD, Lx = eval_model(agent, train_set, calc_ToD, verbose=True)
vToD, vLx = eval_model(agent, dev_set, calc_ToD, verbose=True)
print("{} | ToD = {}  Lx = {} ; vToD = {}  vLx = {}".format(-1, ToD, Lx, vToD, vLx))

sim_start_time = time.time()
########################################################################
for i in range(num_iter): # for each training iteration/epoch
    ToD = 0.0 # estimated ToD over an epoch (or whole dataset)
    Lx = 0.0 # estimated total loss over epoch (or whole dataset)
    n_s = 0
    # run single epoch/pass/iteration through dataset
    ####################################################################
    for batch in train_set:
        n_s += batch[0][1].shape[0] # track num samples seen so far
        x_name, x = batch[0]
        x_hat = agent.settle(x) # conduct iterative inference
        ToD_t = calc_ToD(agent) # calc ToD
        Lx = tf.reduce_sum( metric.bce(x_hat, x) ) + Lx
        # update synaptic parameters given current model internal state
        delta = agent.calc_updates()
        opt.apply_gradients(zip(delta, agent.ngc_model.theta))
        agent.ngc_model.apply_constraints()
        agent.clear()

        ToD = ToD_t + ToD
        print("\r train.ToD {0}  Lx {1}  with {2} samples seen...".format(
              (ToD/(n_s * 1.0)), (Lx/(n_s * 1.0)), n_s),
              end=""
              )
    ####################################################################
    print()
    ToD = ToD / (n_s * 1.0)
    Lx = Lx / (n_s * 1.0)
    # evaluate generalization ability on dev set
```

(continues on next page)

(continued from previous page)

```
    vToD, vLx = eval_model(agent, dev_set, calc_ToD)
    print("---------------------------------------------------")
    print("{} | ToD = {}  Lx = {} ; vToD = {}  vLx = {}".format(
          i, ToD, Lx, vToD, vLx)
          )
```

The above code block represents the core training process loop but you will also find in `sim_train.py` a few other mechanisms that allow for:

1) model saving/check-pointing,
2) an early-stopping mechanism based on patience, and
3) some metric/ToD tracking by storing and saving updated sets of scalar lists to disk. Taking all of the above together, you can simulate the NGC training process after setting some chosen values in your `*.cfg` configuration file, which is read in near the beginning of your training script. For example, in `train_sim.py`, you will see some basic code that reads in an external experimental configuration `*.cfg` file:

```
options, remainder = getopt.getopt(sys.argv[1:], '', ["config=","gpu_id=","n_trials="])
# GPU arguments and configuration
cfg_fname = None
use_gpu = False
n_trials = 1
gpu_id = -1
for opt, arg in options:
    if opt in ("--config"):
        cfg_fname = arg.strip()
    elif opt in ("--gpu_id"):
        gpu_id = int(arg.strip())
        use_gpu = True
    elif opt in ("--n_trials"):
        n_trials = int(arg.strip())
mid = gpu_id
if mid >= 0:
    print(" > Using GPU ID {0}".format(mid))
    os.environ["CUDA_VISIBLE_DEVICES"]="{0}".format(mid)
    #gpu_tag = '/GPU:0'
    gpu_tag = '/GPU:0'
else:
    os.environ["CUDA_VISIBLE_DEVICES"]="-1"
    gpu_tag = '/CPU:0'

save_marker = 1

# load in and build the configuration object
args = Config(cfg_fname) # contains arguments for the simulation
```

Furthermore, notice that the above code snippet contains a bit of setup to allow you to switch to a GPU of your choice (if you set `gpu_id` to a value `>= 0`) or a CPU if no GPU is available (`gpu_id` should be set to `-1`). Notice that the `Config` object reads in a `/path/to/file_name.cfg` text file and produces a queryable object that is backed by a dictionary/hash-table.

The above code blocks/snippets can be found in `train_sim.py` which has been written for you to study and use/run along with the provided example configuration scripts (notice for each model there one subfolder in `/walkthroughs/demo1/` for each of the three models for you to train, each with their own training script `fit.cfg` and `analysis.cfg` script). Let us go ahead and train one of each the three models that we imported into our training script at the start of this demonstration. Run the following three commands as shown below:

```
$ python sim_train.py --config==gncn_t1/fit.cfg --gpu_id=0 --n_trials=1
```

```
$ python sim_train.py --config==gncn_t1_sigma/fit.cfg --gpu_id=0 --n_trials=1
```

```
$ python sim_train.py --config==gncn_pdh/fit.cfg --gpu_id=0 --n_trials=1
```

Alternatively, you can also just run the global bash script `exec_experiments.sh` that we have also provided in `/walkthroughs/demo1/`, which will just simply execute the above three experiments sequentially for you. Run this bash script like so:

```
$ ./exec_experiments.sh
```

As an individual script runs, you will see printed to the terminal, after each epoch, an estimate of the ToD and the BCE over the full training sample as well as the measured validation ToD and BCE over the full development data subset. Note that our training script retrieves from the `walkthroughs/data/mnist/` folder (that you unzipped earlier) only the training arrays, i.e., `trainX.npy` and `trainY.npy`, and the validation set arrays, i.e., `validX.npy` and `validY.npy`. We will use the test set arrays `testX.npy` and `testY.npy` in a follow-up analysis once we have trained each of our models above. After all the processes/scripts terminate, you can check inside each of the model folders, i.e., `walkthroughs/demo1/gncn_t1/`, `walkthroughs/demo1/gncn_t1_sigma/`, and `walkthroughs/demo1/gncn_pdh/`, and see that your script(s) saved/serialized to disk a few useful files:

- `Lx0.npy`: the BCE training loss for the training set over epoch
- `ToD0.npy`: the ToD measurement for the training set over epoch
- `vLx0.npy`: the validation BCE loss over epoch
- `vToD0.npy`: the validation ToD measurement loss over epoch
- `model0.ngc`: your saved/serialized NGC model (with best validation performance)

You can use then plot the numpy arrays using `matplotlib` or your favorite visualization library/package to create curves for each measurement over epoch. The final object, the model object `model0.ngc`, is what we will use in the next section to quantitatively evaluate how well our NGC models work as a generative models.

4.2 Analyzing a Trained Generative Model

Now that you have trained three NGC generative models, we now want to analyze them a bit further, beyond just the total discrepancy and binary cross entropy (the latter of which just tells you how good the model is at auto-associative reconstruction of samples of binary-valued data).

In particular, we are interested in measuring the marginal log likelihood of our models, or `log p(x)`. In general, calculating such a quantity exactly is intractable for any reasonably-sized model since we would have marginalize out the latent variables `Z = {z1, z2, z3}` of each our generative models, requiring us to evaluate an integral over a continuous space. However, though things seem bleak, we can approximate this marginal by resorting to a Monte Carlo estimate and simply draw as many samples as we need (or can computationally handle) from the underlying prior distribution inherent to our NGC model in order to calculate `log p(x)`. Since the NGC models that we have designed/trained in this demo embody an underlying directed generative model, i.e., `p(x,Z) = p(x|z1) p(z1|z2) p(z2|z3) p(z3)`, we can use efficient ancestral sampling to produce fantasy image samples after we query the underlying latent prior `p(z3)`.

Unfortunately, unlike models such as the variational autoencoder (VAE), we do not have an explicitly defined prior distribution, such as a standard multivariate Gaussian, that makes later sampling simple (the VAE is trained with an encoder that forces the generative model to stick as close as it can to this imposed prior). An NGC model's latent prior is, in contrast to the VAE, multimodal and thus simply using a standard Gaussian will not quite produce the fantasized samples we would expect. Nevertheless, we can, in fact, far more accurately capture an NGC model's prior `p(z3)` by treating it as a mixture of Gaussians and instead estimate its multimodal density with a Gaussian mixture model (GMM). Once we have this learned GMM prior, we can sample from this model of `p(z3)` and run these samples through the NGC graph via ancestral sampling (using the prebuilt ancestral projection function `project()`).

ngc-learn is designed to offer some basic support for density estimation, and for this demonstration, we will import and use its GMM density estimator (which builds on top of scikit-learn's GMM base model), i.e., `ngclearn.density.gmm`. First, we will need to extract the latent variables from the trained NGC model, which simply requires us to adapt our `eval_model()` function in our training script to also now return a design matrix where each row contains one latent code vector produced by our model per data point in a sample pool. Specifically, all we need to write is the following:

```
def extract_latents(agent, dataset, calc_ToD, verbose=False):
    """
        Extracts latent activities of an agent on a fixed-point data sample
    """
    latents = None
    ToD = 0.0
    Lx = 0.0
    N = 0.0
    for batch in dataset:
        x_name, x = batch[0]
        N += x.shape[0]
        x_hat = agent.settle(x) # conduct iterative inference
        lats = agent.ngc_model.extract(node_name, cmpt_name)
        if latents is not None:
            latents = tf.concat([latents,lats],axis=0)
        else:
            latents = lats
        ToD_t = calc_ToD(agent) # calc ToD
        # update tracked fixed-point losses
        Lx = tf.reduce_sum( metric.bce(x_hat, x) ) + Lx
        ToD = calc_ToD(agent) + ToD
        agent.clear()
        print("\r ToD {0}  Lx {1} over {2} samples...".format((ToD/(N * 1.0)), (Lx/(N *␣
↪1.0)), N),end="")
    print()
    Lx = Lx / N
    ToD = ToD / N
    return latents, ToD, Lx
```

Notice we still keep our measurement of the ToD and BCE just as an extra sanity check to make sure that any model we de-serialize from disk yields values similar to what we measured during our training process. Armed with the extraction function above, we can gather the latent codes of our NGC model. Notice that in the provided `walkthroughs/demo1/extract_latents.py` script, you will find the above function fully integrated and used. Go ahead and run the extraction script for the first of your three models:

```
$ python extract_latents.py --config==gncn_t1/analyze.cfg --gpu_id=0
```

and you will now find inside the folder `walkthroughs/demo1/gncn_t1/` a new numpy array file `z3_0.npy`, which contains all of the latent variables for the top-most layer of your `GNCN-t1` model (you can examine the configuration file `analyze.cfg` to see what arguments we set to achieve this).

Now it is time to fit the GMM prior. In the `fit_gmm.py` script, we have set up the necessary framework for you to do so (using `18,000` samples from the training set to speed up calculations a bit). All you need to do at this point, still using the `analyze.cfg` configuration, is execute this script like so:

```
$ python fit_gmm.py --config==gncn_t1/analyze.cfg --gpu_id=0
```

and after your fitting process script terminates, you will see inside your model directory `walkthroughs/demo1/gncn_t1/` that you have a de-serialized learned prior `prior0.gmm`.

With this prior model `prior0.gmm` and your previously trained NGC system `model0.ngc`, you are ready to finally estimate your marginal log likelihood `log p(x)`. The final script provided for you, i.e., `walkthroughs/demo1/eval_logpx.py`, will do this for you. It simply takes your full system – the prior and the model – and calculates a Monte Carlo estimate of its log likelihood using the test set. Run this script as follows:

```
$ python eval_logpx.py --config==gncn_t1/analyze.cfg --gpu_id=0
```

and after it completes (this step can take a bit more time than the other steps, since we are computing our estimate over quite a few samples), in addition to outputting to I/O the calculated `log p(x)`, you will see two more items in your model folder `walkthroughs/demo1/gncn_t1/`:

- `logpx_results.txt`: the recorded marginal log likelihood
- `samples.png`: some visual samples stored in an image array for you to view/assess

If you `cat` the first item, you should something similar to the following (which should be the same as what was printed to I/O when you evaluation script finished):

```
$ cat gncn_t1/logpx_results.txt
Likelihood Test:
  log[p(x)] = -103.63043212890625
```

and if you open and view the image samples, you should see something similar to:

Now go ahead and re-run the same steps above but for your other two models, using the final provided configuration scripts, i.e., `gncn_t1_sigma/analyze.cfg` and `gncn_pdh/analyze.cfg`. You should see similar outputs as below.

For the GNCN-t1-Sigma, you get a log likelihood of:

```
$ cat gncn_t1_sigma/logpx_results.txt
Likelihood Test:
  log[p(x)] = -99.73319244384766
```

with images as follows:

For the GNCN-PDH, you get a log likelihood of:

```
$ cat gncn_t1_sigma/logpx_results.txt
Likelihood Test:
  log[p(x)] = -97.39334106445312
```

with images as follows:

For the three models above, we get log likelihood measurements that are desirably within the right ballpack of those reported in related literature [3].

You have now successfully trained three different, powerful NGC generative models using ngc-learn's Model Museum and analyzed their log likelihoods. While these models work/perform well, there is certainly great potential for further improvement, extensions, and alternative ideas and it is our hope that future research developments will take advantage of the tools/framework provided by ngc-learn to produce even better results.

Note: Setting the `--n_trials` flag to a number greater than one for the above training scripts to run each simulation for multiple, different experimental trials (each set of files/objects will be indexed by its trial number).

4.3 References

[1] Rao, Rajesh PN, and Dana H. Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects." Nature Neuroscience 2.1 (1999): 79-87. [2] Friston, Karl. "Hierarchical models in the brain." PLoS Computational Biology 4.11 (2008): e1000211. [3] Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022).

CHAPTER FIVE

WALKTHROUGH 2: CREATING CUSTOM NGC PREDICTIVE CODING SYSTEMS

In this demonstration, we will learn how to craft our own custom NGC system using ngc-learn's fundamental building blocks – nodes and cables. After going through this demonstration, you will:

1. Be familiar with some of ngc-learn's basic theoretical motivations.
2. Understand ngc-learn's basic building blocks, nodes and cables, and how they relate to each other and how they are put together in code. Furthermore, you will learn how to place these connected building blocks into a simulation object to implement inference and learning.
3. Craft and simulate a custom nonlinear NGC model based on exponential linear units to learn how to mimic a streaming mixture-based data generating process. In this step, you will learn how to design an ancestral projection graph to aid in fantasizing data patterns that look like the target data generating process.

Note that the folder of interest to this demonstration is:

- `walkthroughs/demo2/`: this contains the necessary simulation script

5.1 Theoretical Motivation: Nodes, Compartments, and Cables

At its core, part of ngc-learn's fundamental design is inspired by (neural) cable theory , where neurons, arranged in complex connectivity structures, are viewed as performing dendritic calculations. In other words, a particular neuron integrates information from different input signals (for example, those from other neurons), in often highly nonlinear ways through a complex dendritic tree.

Although modeling a neuronal system through the lens of cable theory is certainly complex and intricate in of itself, ngc-learn is built in this direction, starting with the idea a neuron (or a cluster of them) can be viewed as a node, or *Node* (also see *Node Model*), and each bundle of synapses that connect nodes can be viewed as a cable, or *Cable* (also see *Cable Model*). Each node has different, multiple "compartments" (which are named), which are regions or slots inside the node that other nodes can deposit information/signals into. These compartments allow a node to collect information from many different connected/related nodes and then, within its integration routine (or `step()`), decide how to combine the different signals in order to calculate its own activity (loosely corresponding to a rate-coded firing rate – we will learn how to model spike trains in a later demonstration). As a result, many nodes and cables yield an NGC system where each node is itself, in general, a stateful computation (even if we are processing static data such as images).

5.2 Building and Simulating NGC Systems

5.2.1 The Building Blocks: Nodes and Cables

With the above aspect of ngc-learn's theoretical framing in mind, we can craft connectivity patterns of our own by deciding the form that each node and cable in our system will take. ngc-learn currently offers a few core nodes and cable types (note ngc-learn is an evolving software framework, so more node/cable types are to come in future releases, either through the NAC team or community contributions). The core node type set currently includes `SNode`, `ENode`, and `FNode` (all inheriting from the `Node` base class) while the current cable type set includes `DCable` and `SCable` (all inherited from the `Cable` base class).

An `SNode` refers to a stateful node (see *SNode*), which is one of the primary nodes you will work with when crafting NGC systems. A stateful node contains inside of it a cluster (or block) of neurons, the number of which is controlled through the `dim` argument. To initialize a state node, we simply invoke the following:

```
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
prior_cfg = {"prior_type" : "laplace", "lambda" : 0.001}
a = SNode(name="a", dim=64, beta=0.1, leak=0.001, act_fx="relu",
           integrate_kernel=integrate_cfg, prior_kernel=prior_cfg)
```

where we notice that the above creates a state node with `64` neurons that will update themselves according to an Euler integration step (step size of `0.1` and a leak of `0.001`) and apply a `relu` post-activation to compute their post-activity values. Furthermore, notice that a Laplacian prior has been place over the neural activities within the state `a` (weighted by the strength coefficient `lambda`) – such a prior is meant to encourage neural activity values towards zero (yielding sparser patterns). A state node, in ngc-learn 0.0.1, contains five key compartments: `dz_td`, `dz_bu`, `z`, `phi(z)`, and `mask`. `z` represents the actual state values of the neurons inside the node while the compartment `phi(z)` is the nonlinear transform of `z` (indicating the application of the node's encoded activation/transfer function, e.g., `relu` in the case of node `a` in the example above). `dz_td` and `dz_bu` are state update compartments, where (vector) signals from other nodes are deposited (and summed together vector-wise), with the notable exception that `dz_bu` can be weighted by the first derivative of the activation function encoded into the state node (for example, in `a` above, signals deposited into `dz_bu` are element-wise multiplied by the `relu` derivative, or `d.phi(z)/d.z = d.relu(z)/d.z`). While, in principle, any node can be made to deposit into any compartment of another node, the intentional and primary use of an `SNode` entails letting the node itself automatically update `z` and `phi(z)` according to the integration function configured (such as Euler integration) while letting other nodes deposit signal values into `dz_td` and `dz_bu`. (This demonstration will assume this form of operation.)

While a state node by itself is not all that interesting, when we connect it to another node, we create a basic computation system where signals are passed from a source node to a destination node. To connect a node to another node, we need to wire them together with a `Cable`, which can transform signals between them with a dense bundle of synapses (as in the case of a `DCable`) or simply carry along and potentially weight by a fixed scalar multiplication (as in the case of an `SCable`). For example, if we want to wire node `a` to a node `b` through a dense bundle of synapses, we would do the following:

```
a = SNode(name="a", dim=64, beta=0.1, leak=0.001, act_fx="relu",
          integrate_kernel=integrate_cfg, prior_kernel=prior_cfg)
b = SNode(name="b", dim=32, beta=0.05, leak=0.002, act_fx="identity",
          integrate_kernel=integrate_cfg, prior_kernel=None)

init_kernels = {"A_init" : ("gaussian",0.025)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}
a_b = a.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
```

where we note that the cable/wire `a_b`, of type `DCable` (see *DCable*), will pull a signal from the `phi(z)` compartment of node `a` and transmit/transform this signal along the synaptic parameters it embodies (a dense matrix where each synaptic value is randomly initialized from a zero-mean Gaussian distribution and standard deviation of `0.025`) and place the resultant signal inside the `dz_td` compartment of node `b`.

Currently, an `SNode` (in ngc-learn version 0.2.0), integrates over two compartments – `dz_td` (top-down pressure signals) and `dz_bu` (bottom-up potentially weighted signals), and finally combines them through a linear combination to produce a full update to the internal state compartment `z`. Note that many external nodes can deposit signal values into each compartment `dz_td` and `dz_bu` and each new deposit value is directly summed with the current value of the compartment. For example, a five-node system/circuit could take the following form:

```
carryover_cable_cfg = {"type": "simple", "coeff": 1.0} # an identity cable
a = SNode(name="a", dim=10, beta=0.1, leak=0.001, act_fx="identity")
b = SNode(name="b", dim=5, beta=0.05, leak=0.002, act_fx="identity")
c = SNode(name="c", dim=2, beta=0.05, leak=0.0, act_fx="identity")
d = SNode(name="d", dim=2, beta=0.05, leak=0.0, act_fx="identity")
e = SNode(name="e", dim=15, beta=0.05, leak=0.0, act_fx="identity")

a_c = a.wire_to(c, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
b_c = b.wire_to(c, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
d_c = d.wire_to(c, src_comp="phi(z)", dest_comp="dz_bu", cable_kernel=carryover_cable_
↪cfg)
e_c = e.wire_to(c, src_comp="phi(z)", dest_comp="dz_bu", cable_kernel=dcable_cfg)
```

where `a` and `b` both deposit signals (which will be summed) into the `dz_td` compartment of `c` while `d` and `e` deposit signals into the `dz_bu` compartment of `c`. Crucially notice that we introduce the other type of cable from `d` to `c`, i.e., an `SCable` (see *SCable*), which is a simple carry-over cable that we have merely configured (in the dictionary `carryover_cable_cfg`) to only pass along information information from node `d` to `c`, simply multiplying the vector by `1.0` (NOTE: if a simple cable is being used, the dimensionality of the source node and the destination node should be exactly the same). Bear in mind that any general `Cable` object is directional – it only transmits in the direction of its set wiring pattern (from `src_comp` of its source node to the `dest_comp` of the destination node). So if it is desired, for instance, that information flows not only from `a` to `c` but from `c` to `a`, then one would need to directly wire node `c` back to `a` following a similar pattern as in the code snippet above. Finally, note that when you wire together two nodes, they each become aware of this wiring relationship (i.e., node `a` understands that it feeds into node `c` and node `c` knows that `a` feeds into it).

To learn or adjust the synaptic cables connecting the nodes in the five-node system we created above, we need to configure the cables themselves to use a local Hebbian-like update. For example, if we want the cable `a_c` to evolve over time, we notify the node that it needs to update according to:

```
a_c.set_update_rule(preact=(a,"phi(z)"), postact=(c,"phi(z)"), param=["A"])
```

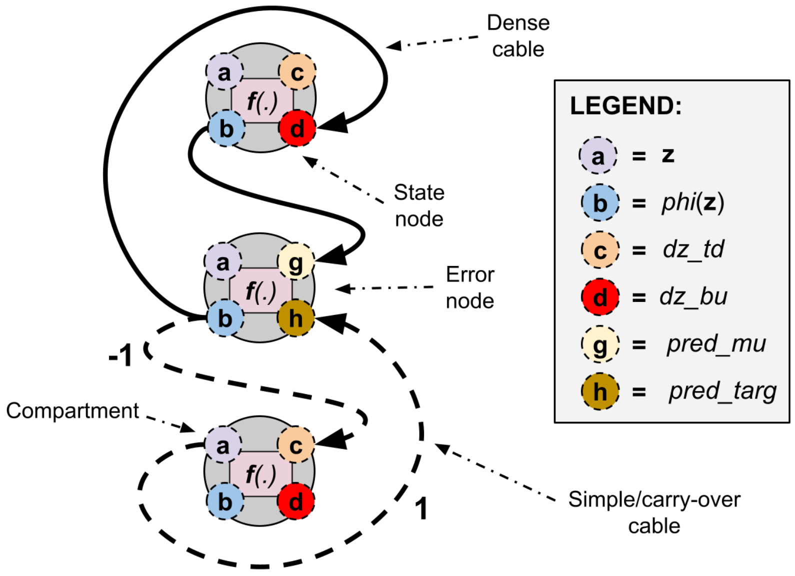
where the above sets a (two-factor) Hebbian update that will compute an adjustment matrix of the same shape as the underlying synaptic matrix that connects `a` to `c` (essentially a product of post-activation values in `a` with post-activation values in `c`). Notice that a pre-activation term (`preact`) requires a 2-tuple containing a target node object and a string denoting which compartment within that node to extract information from to create the pre-synaptic Hebbian term. (`postact` refers to the post-activation term, the argument itself following the same format at `preact`).

Beyond the `SNode`, we need to study one more important type of node – the `ENode` (see *ENode*). While, in principle, one could build a complete NGC system with just state nodes and cables (which will be the subject of future walk-throughs/tutorials), an important aspect of NGC computation we have not addressed is that of the `error neuron`, represented in ngc-learn by an `ENode`. An `ENode` is a special type of node that performs a mismatch calculation (or a computation that compares how far off one quantity is from another) and is, in fact, a mathematical simplification of a state node known as a fixed-point. In short, one can simulate a mismatch calculation over time by simply modeling the final result such as the (vector) subtraction of one value from another. In ngc-learn (up and including version 0.2.0), in addition to `z` and `phi(z)`, the `ENode` also contains the key following compartments: `pred_mu`, `pred_targ`, and `L`. `pred_mu` is a compartment that contains a summation of deposits that represent an external signals that form a "prediction" (or expectation) while `pred_targ` is a compartment that contains a summation of external signals that form a "target" (or desired value/signal). L is a useful compartment as this is internally calculated by the error node to represent the loss function by which the fixed-point calculation is derived, i.e., in the case of simple subtraction where `pred_mu - pred_targ`, this would mean that the error node is calculating the first derivative of the mean squared error (MSE).

Now that we know how an error node works, let us create a simple 3-node circuit that leverages an error node mismatch computation:

```
init_kernels = {"A_init" : ("gaussian",0.025)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}
pos_carryover = {"type": "simple", "coeff": 1.0}
neg_carryover = {"type": "simple", "coeff": -1.0}

# Notice that we make b and e have the same dimension (10) given that we
# want to wire their information exchange paths with SCable(s)

a = SNode(name="a", dim=20, beta=0.05, leak=0.001, act_fx="identity")
b = SNode(name="b", dim=10, beta=0.05, leak=0.002, act_fx="identity")
e = ENode(name="e", dim=10)

# wire the states a and b to error neurons/node e
a_e = a.wire_to(e, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=dcable_cfg)
b_e = b.wire_to(e, src_comp="z", dest_comp="pred_targ", cable_kernel=pos_carryover)

# wire error node e back to nodes a and b to provide feedback to their states
e.wire_to(a, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(a_e,"A^T"))
e.wire_to(b, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_carryover)

# set up local Hebbian updates for a_e
a_e.set_update_rule(preact=(a,"phi(z)"), postact=(e,"phi(z)"), param=["A"])
```

where we see that node `a` deposits a prediction signal into the `pred_mu` compartment of `e` and node `b` deposits a target signal into the `pred_targ` compartment of `e` (where a simple cable `pos_carryover` will just multiply this signal by 1 and dump it into the appropriate compartment). Notice that we have wired `e` back to `a` using a special flag/argument in the `wire_to()` routine, i.e., `mirror_path_kernel`. This special argument simply takes in a 2-tuple where the first element is the physical cable object we want to reuse while the second is a string flag telling ngc-learn how to re-use the cable (in this case, `A^T`, which means that we use the transpose of the underlying weight matrix contained inside of the dense cable `a_e`). Also observe that `e` has been wired back to node `b` with a simple cable that multiplies the post-activation of `e` by `-1`. The above 3-node circuit we have built is illustrated in the diagram below.



Before we turn our attention to simulating the interactions/processing of the above nodes and cables, there is one more specialized node worthy of mention – the forward node or `FNode` (see *FNode*). This node is simple – it only contains three compartments: `dz`, `z`, and `phi(z)`. An `FNode` operates much like an `SNode` except that it fundamentally is "stateless" – external nodes deposit signals into `dz` (where multiple deposits are vector summed) and then this value is directly and automatically placed inside of `z` after which an encoded activation function is applied to compute `phi(z)`. Note that an `SNode` can be modified to also behave like an `FNode` by setting its argument `.zeta` (or the amount of recurrent carry-over inside the neural state dynamics) equal to `0` and setting `beta` to `1`. However, the `FNode` is a convenience node and is often used to build an ancestral projection graph, of which we will describe later.

5.2.2 Simulating Connected Nodes as Systems with the NGCGraph

Now that we have a basic grasp as to how nodes and cables can be instantiated and connected to build neural circuits, let us examine the final key step required to build an NGC system – the simulation object `NGCGraph` (see *NGCGraph*).

An `NGCGraph` is a general structure that will take in nodes that have been wired together with cables and simulate their evolution/processing over time. This structure crucially allows us to specify the execution sequence of nodes (or order of operations) within a discrete step of simulation time. It also provides several basic utility functions to facilitate analysis of the internal nodes. In this demo, we will focus on the core primary low-level routines one will want to conduct most simulations, i.e., `set_cycle()`, `settle()`, `apply_constraints()`, `calc_updates()`, `clear()`, and `extract()`. (Note that higher-level convenience functions that combine all of these functions together, like `evolve()`, could be used, but we will not cover them in this demonstration.) Let us take the five node circuit we built earlier and place them in a system simulation:

```
model = NGCGraph(K=5)
model.proj_update_mag = -1.0 # bound the calculated synaptic updates (<= 0 turns this␣
↪off)
model.proj_weight_mag = 1.0 # constrain the Euclidean norm of the rows of each synaptic␣
↪matrix
model.set_cycle(nodes=[a,b,c,d]) # execute nodes a through d (in order left to right)
model.set_cycle(nodes=[e]) # execute node e
model.apply_constraints() # immediately applies constraints to synapses after␣
↪initialization
model.compile(batch_size=1)
```

where the above seven lines of code create a full NGC system using the nodes and cables we set before. The set_cycle() function takes in a list/array of nodes and tells the underlying NGCGraph system to execute them (in the order of their appearance within the list) first at each step in time. Making multiple subsequent calls to set_cycle() will add in addition execution cycles to an NGC system's step. Note that one simulation step of an NGCGraph consists of multiple cycles, executed in the order of their calls when the simulation object was initialized. For example, one step of our "model" object above would first execute the internal .step() functions of a, b, c, then d in the first cycle and then execute the .step() of e in the second cycle. Also observe that in our NGCGraph constructor, we have told ngc-learn that simulations are only ever to be K=5 discrete time steps long. Finally note that, when you set execution cycles for an NGCGraph, ngc-learn will examine the cables you wired between nodes and extract any learnable synaptic weight matrices into a parameter object .theta.

The final item to observe in the code snippet above is the call to compile() routine. This function is run after putting together your NGCGraph in order to ensure the entire system is self-coherent and set up to work correctly with the underlying static graph compilation used by Tensorflow 2 to drastically speed up your code (Note: the compile() routine and static graph optimization was integrated into ngc-learn version 0.2.0 onward.) The only argument you need to set for compile() is the batch_size argument – you must decide what fixed batch size you will use throughout simulation so that way ngc-learn can properly compile a static graph in order to optimize the underlying code for fast in-place memory calculations and other computation graph specific items. Note that if you do not wish to use ngc-learn's static graph optimization, simply set the use_graph_optim to False via .compile(use_graph_optim=False), which will allow you to use variable-length batch sizes (at the cost of a bit slower computation).

With the above code, we are now done building the NGC system and can begin using it to process and adapt to sensory data. To make our five-node circuit process and learn from a single data pattern, we would then write the following:

```
opt = # ... set some TF optimization algorithm, such as SGD, here ...
x = tf.ones([1,10])
readouts = model.settle(
                clamped_vars=[("c","z",x)],
                readout_vars=[("a","phi(z)"),("b","phi(z)"),("d","phi(z)"),("e","phi(z)
↪")]
            )
print("The value of {} w/in Node {} is {}".format(readouts[0][0], readouts[0][1],␣
↪readouts[0][2].numpy()))
# update synaptic parameters given current model internal state
delta = model.calc_updates()
opt.apply_gradients(zip(delta, model.theta)) # apply a TF optimizer here
model.apply_constraints()
model.clear() # reset the underlying node states back to resting values
```

where we have crafted a trivial example of processing a vector of ones (x), clamping this value to node c's compartment z (note that clamping means we fix the node's compartment to a specific value and never let it evolve throughout simulation), and then read out the value of the phi(v) compartment of nodes a, b, c, and e. The readout_vars argument to settle() allows us to tell an NGCGraph which nodes and which compartments we want to observe after it run its simulated settling process over K=5 steps. An NGCGraph saves the output of settle() into the readouts variable which is a list of triplets of the form [(node_name, comp_name, value),...] and, in the example, above we are deciding to print out the first node's value (in its set phi(z) compartment). After the NGCGraph executes its settling process, we can then tell it to update all learnable synaptic weights (only for those cables that were configured to use a Hebbian update with set_update_rule()) via the calc_updates(), which itself returns a list of the synaptic weight adjustments, in the order of the synaptic matrices the NGCGraph object placed inside of .theta.

Desirably, after you have obtained delta from calc_updates(), you can then use it with a standard Tensorflow 2 adaptive learning rate such as stochastic gradient descent or Adam. An important point to understand is that an NGC system attempts to maximize its total discrepancy, which is a negative quantity that it would like to be at zero (meaning all local losses within it have reached an equilibrium at zero) – this is akin to optimizing the approximate free energy of the system. Internally, an NGCGraph will multiply the Hebbian updates by a negative coefficient to allow the user to directly use an external minimizer from a library such as Tensorflow.

After updating synaptic matrices using a Tensorflow optimizer, one then calls `apply_constraints()` to ensure any weight matrix constraints are applied after updating, finally ending with a call to `clear()`, which resets the values of all nodes in the `NGCGraph` back to zero (or a resting state). (Note that if you do not want the NGC system to reset its internal nodes back to resting zero states, then simply do not call `clear()` – for example, on a long temporal data stream such as a video feed, you might not want to reset the NGC system back to its zero-resting state until the video clip terminates).

5.3 Learning a Data Generating Process: A Streaming NGC Model

Now that we familiarized ourselves with the basic mechanics of nodes and cables as well as how they fit within a simulation graph, let us apply our knowledge to build a nonlinear NGC generative model that learns to mimic a streaming data generating process. Note that this part of the demonstration corresponds to the materials/scripts provided within `walkthroughs/demo2/`.

In ngc-learn, within the `generator` module, there are a few data generators to facilitate prototyping and simulation studies. Simulated data generating processes can be used in lieu of real datasets and are useful for early preliminary experimentation and proof-of-concept demonstrations (in statistics, such experiments are called "simulation studies"). In this demonstration, we will take a look at the `MoG` (mixture of Gaussians, see *MoG*) static data generating process. Data generating processes in ngc-learn typically offer a method called `sample()` and, depending on the type of process being used, with process-specific arguments. In the `MoG` process, we can initialize a non-temporal (thus "static") process as follows:

```
# ...initialize mu1, mu2, mu3, cov1, cov2, cov3...
mu_list = [mu1, mu2, mu3]
sigma_list = [cov1, cov2, cov3]
process = MoG(means=mu_list, covar=sigma_list, seed=69)
```

where the above creates a fixed mixture model of three multivariate Gaussian distributions (each component has an equal probability of being sampled by default in the `MoG` object). In the demonstration script `sim_dyn_train.py`, you can see what specific mean and covariance values we have chosen (for simplicity, we set our problem space to be two-dimensional and have each covariance matrix designed to be explicitly diagonal). The advantage of a data generator that we will exploit in this demonstration is the fact that it can be queried online, i.e., we can call its `sample()` function to produce fresh data sampled from its underlying generative process. This will allow
us to emulate the scenario of training an NGC system on a data stream (as opposed to a fixed dataset like we did in the first demonstration).

With the data generating process chosen and initialized, we now turn to our NGC generative model. The model we will construct will be a nonlinear model with three layers – a sensory layer `z0` and two latent neural variable layers `z1` and `z2`. The post-activation for `z1` will be the exponential linear rectifier unit (ELU) while the second layer will be set to the identity and bottle-necked to a two-dimensional code so we can visualize the top-most latents easily later. Our goal will be train our NGC model for several iterations and then use it to synthesize/fantasize a new pool of samples, one for each known component of our mixture model (since each component represents a "label") where we will finally estimate the sample mean and covariance of each particular pool to gauge how well the model has been fit to the mixture process.

We create the desired NGC model as follows:

```
batch_size = 32
# create cable wiring scheme relating nodes to one another
wght_sd = 0.025 #0.025 #0.05
init_kernels = {"A_init" : ("gaussian",wght_sd)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 69}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}
```

(continues on next page)

(continued from previous page)

```
neg_scable_cfg = {"type": "simple", "coeff": -1.0}
constraint_cfg = {"clip_type":"norm_clip","clip_mag":1.0,"clip_axis":1}

z2_mu1 = z2.wire_to(mu1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z2_mu1.set_constraint(constraint_cfg)
mu1.wire_to(e1, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z1.wire_to(e1, src_comp="z", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e1.wire_to(z2, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z2_mu1,"A^T"))
e1.wire_to(z1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

z1_mu0 = z1.wire_to(mu0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z1_mu0.set_constraint(constraint_cfg)
mu0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z1_mu0,"A^T"))
e0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

# set up update rules and make relevant edges aware of these
z2_mu1.set_update_rule(preact=(z2,"phi(z)"), postact=(e1,"phi(z)"), param=["A"])
z1_mu0.set_update_rule(preact=(z1,"phi(z)"), postact=(e0,"phi(z)"), param=["A"])

# Set up graph - execution cycle/order
model = NGCGraph(K=K)
model.set_cycle(nodes=[z2,z1,z0])
model.set_cycle(nodes=[mu1,mu0])
model.set_cycle(nodes=[e1,e0])
model.apply_constraints()
model.compile(batch_size=batch_size)
```

which constructs the model the three-layer system, which we can also depict with the following ngc-learn design short-hand:

```
Node Name Structure:
z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
```

One interesting thing to note is that, in the `sim_dyn_train.py` script, we also create an ancestral projection graph (or a co-model) in order to conduct the sampling we want to do after training. An ancestral projection graph `ProjectionGraph` (see *ProjectionGraph*), which is useful for doing things like ancestral sampling from a directed generative model, should generally be created after an `NGCGraph` object has been instantiated. Doing so, as seen in `sim_dyn_train.py`, entails writing the following:

```
# build an ancestral sampling graph
z2_dim = model.getNode("z2").dim
z1_dim = model.getNode("z1").dim
z0_dim = model.getNode("z0").dim
# Set up complementary sampling graph to use in conjunction w/ NGC-graph
s2 = FNode(name="s2", dim=z2_dim, act_fx="identity")
s1 = FNode(name="s1", dim=z1_dim, act_fx="elu")
s0 = FNode(name="s0", dim=z0_dim, act_fx="identity")
s2_s1 = s2.wire_to(s1, src_comp="phi(z)", dest_comp="dz", mirror_path_kernel=(z2_mu1,"A
↪"))
s1_s0 = s1.wire_to(s0, src_comp="phi(z)", dest_comp="dz", mirror_path_kernel=(z1_mu0,"A
↪"))
```

(continues on next page)

(continued from previous page)

```
sampler = ProjectionGraph()
sampler.set_cycle(nodes=[s2,s1,s0])
sampler.compile()
```

Creating a `ProjectionGraph` is rather similar to creating an `NGCGraph` (notice that we chose to use `FNode`(s) since they work well for feedforward projection schemes). However, we should caution that the design of a projection graph should meaningfully mimic what one would envision is the underlying directed, acyclic generative model embodied by their `NGCGraph` (it helps to draw out/visualize the dot-and-arrow structure you want graphically first, using similar shorthand as we presented for our model above, in order to then extract the underlying generative model the system implicitly learns). A few important points we followed for designing the projection graph above:

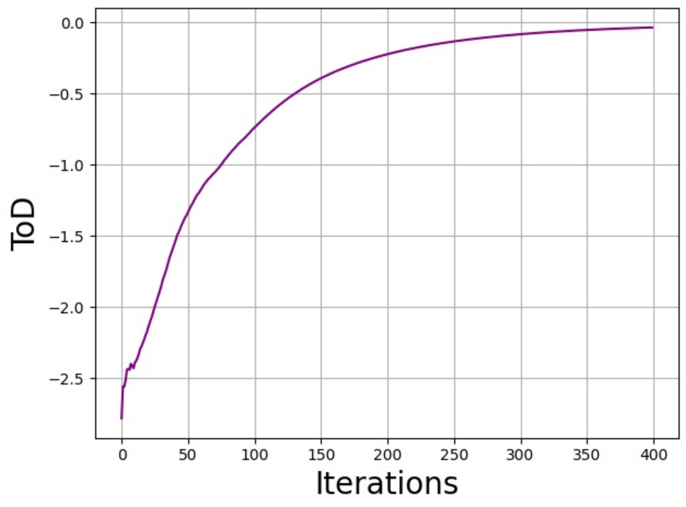
1) the number (dimensionality) of nodes should be the same as the state nodes in the NGC system, i.e., `s2` corresponds to `z2`, `s1` corresponds to `z1`, and `s0` corresponds to `z0`;

2) the cables connecting the nodes should directly share the exact synaptic matrices between each key layer of the original NGC system, i.e., the cable `s2_s1` points directly to/re-uses cable `z2_mu1` and cable `s1_s0` points directly to/re-uses cable `z1_mu0` (note that we use the special argument `A` in the `wire_to()` function that allows directly shallow-copying/linking between relevant cables). Notice that we used another one of the `NGCGraph` utility functions – `getNode()` – which directly extracts a whole `Node` object from the graph, allowing one to quickly call its internal data members such as its dimensionality `.dim`.

With the above `NGCGraph` and `ProjectionGraph` now created, we can now train our model by sampling the `MoG` data generator online as follows:

```
ToD = 0.0
Lx = 0.0
Ns = 0.0
alpha = 0.99 # fading factor
for iter in range(n_iterations):

    x, y = process.sample(n_s=batch_size)
    Ns = x.shape[0] + Ns * alpha

    # conduct iterative inference & update NGC system
    readouts, delta = model.settle(
                        clamped_vars=[("z0","z",x)],
                        readout_vars=[("mu0","phi(z)"),("mu1","phi(z)")]
                      )
    x_hat = readouts[0][2]

    ToD = calc_ToD(model) + ToD * alpha # calc ToD
    Lx = tf.reduce_sum( metric.mse(x_hat, x) ) + Lx * alpha
    # update synaptic parameters given current model internal state
    for p in range(len(delta)):
        delta[p] = delta[p] * (1.0/(x.shape[0] * 1.0))
    opt.apply_gradients(zip(delta, model.theta))
    model.apply_constraints()
    model.clear()

    print("\r{} | ToD = {}  MSE = {}".format(iter, ToD/Ns, Lx/Ns), end="")
print()
```

where we track the total discrepancy (via a custom `calc_ToD()` also written for you in `sim_dyn_train.py`, much as we did in Demonstration # 1) as well as the mean squared error (MSE). Notably, for online streams, we track a particularly useful form of both metrics – prequential MSE and prequential ToD – which are essentially adaptations of the prequential error measurement [1] used to track the online performance of classifiers/regressors on data streams. We will plot the prequential ToD at the end of our simulation script, which will yield a plot that should look similar to (where we see that ToD is maximized over time):



Finally, after training, we will examine how well our NGC system learned to mimic the MoG by using the co-model projection graph we created earlier. This time, our basic process for sampling from the NGC model is simpler than in Demonstration # 1 where we had to learn a density estimator to serve as our model's prior. In this demonstration, we will approximate the modes of our NGC's model's prior by feeding in batches of test samples drawn from the MoG process, about 64 samples per component, running them through the NGCGraph to infer the latent z2 for each sample, estimate the latent mean and covariance for mode, and then use these latent codes to sample from and project through our ProjectionGraph. This will get us our system's fantasized samples from which we can estimate the generative model's mean and covariance for each pool, allowing us visually compare to the actual mean and covariance of each component of the MoG process. This we have done for you in the sample_system() routine, shown below:

```python
def sample_system(Xs, model, sampler, Ns=-1):
    readouts, _ = model.settle(
                    clamped_vars=[("z0","z",tf.cast(Xs,dtype=tf.float32))],
                    readout_vars=[("mu0","phi(z)"),("z2","z")],
                    calc_delta=False
                  )
    z2 = readouts[1][2]
    z = z2
    model.clear()
    # estimate latent mode mean and covariance
    z_mu = tf.reduce_mean(z2, axis=0, keepdims=True)
    z_cov = stat.calc_covariance(z2, mu_=z_mu, bias=False)
    z_R = tf.linalg.cholesky(z_cov) # decompose covariance via Cholesky
    if Ns > 0:
        eps = tf.random.normal([Ns, z2.shape[1]], mean=0.0, stddev=1.0, seed=69)
    else:
        eps = tf.random.normal(z2.shape, mean=0.0, stddev=1.0, seed=69)
    # use the re-parameterization trick to sample this mode
    Zs = z_mu + tf.matmul(eps,z_R)
    # now conduct ancestral sampling through the directed generative model
    readouts = sampler.project(
                    clamped_vars=[("s2","z", Zs)],
```

(continues on next page)

(continued from previous page)

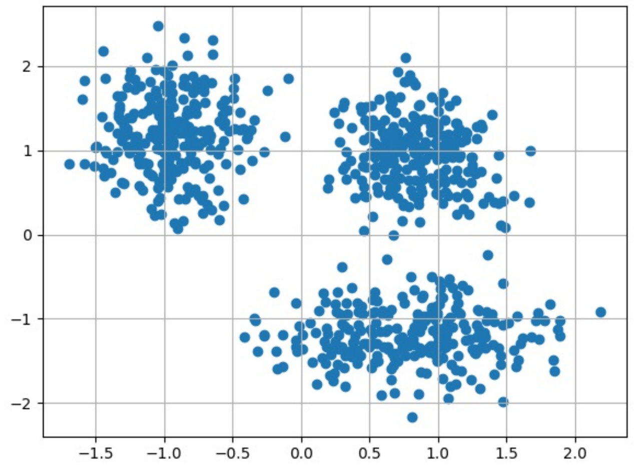
```
                    readout_vars=[("s0","phi(z)")]
                )
    X_hat = readouts[0][2]
    sampler.clear()
    # estimate the mean and covariance of the sensory sample space of this mode
    mu_hat = tf.reduce_mean(X_hat, axis=0, keepdims=True)
    sigma_hat = stat.calc_covariance(X_hat, mu_=mu_hat, bias=False)
    return (X_hat, mu_hat, sigma_hat), (z, z_mu, z_cov)
```

Note inside the `sim_dyn_train.py` script, we have also written several helper functions for plotting the latent variables, input space samples, and the data generator and model-estimated input means/covariances. We have set the number of training iterations to be `400` and the online mini-batch size to be `32` (meaning that we draw `32` samples from the `MoG` each iteration).

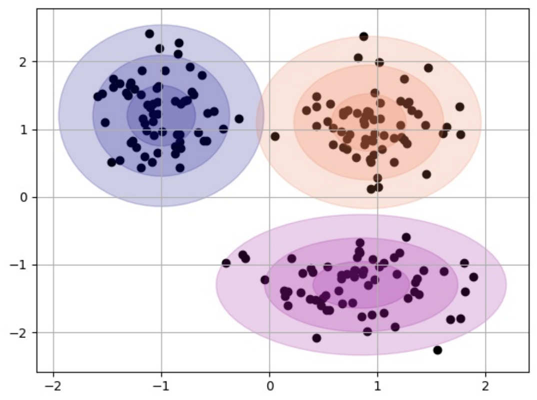
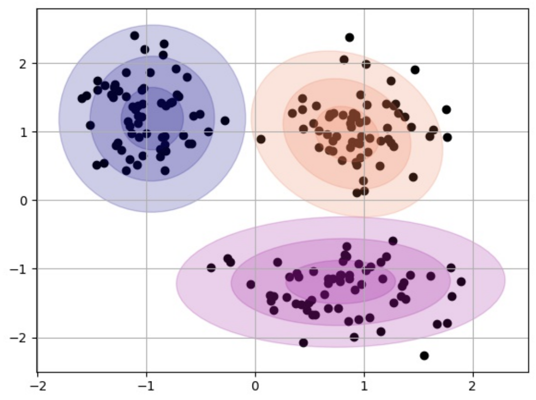
You can now execute the demonstration script as follows:

```
$ python sim_dyn_train.py
```

and you will see that our exponential linear model produces the following samples:

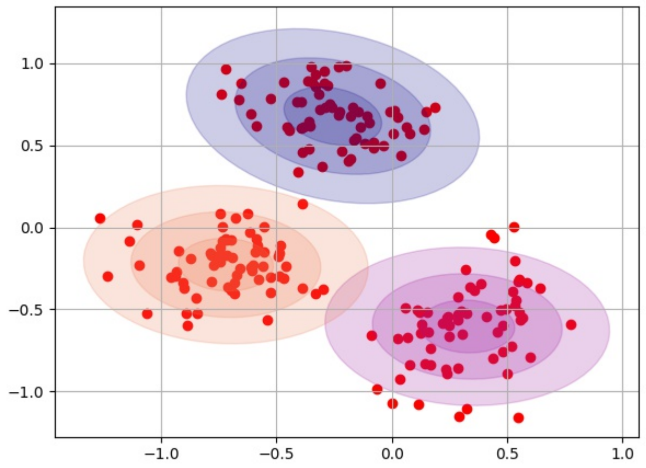


and results in the following fit (Right) as compared to the original `MoG` process (Left):

Original Process	NGC Model Fit
	

We observe that the NGC model does a decent job of learning to mimic the underlying data generating process, although we can see it is not perfect as a few data points are not quite captured within its covariance envelope (notably in the orange Gaussian blob in the top right of the plot).

Finally, we visualize our model's latent space to see how the 2D codes clustered up and obtain the plot below:



Desirably, we observe that our latent codes have clustered together and yielded a sufficiently separable latent space (in other words, the codes result in distinct modes where each mode of the `MoG` is represented a specific blob/grouping in latent space.).

As a result, we have successfully learned to mimic a synthetic mixture of Gaussians data generating process with our custom, nonlinear NGC system.

5.4 References

[1] Gama, Joao, Raquel Sebastiao, and Pedro Pereira Rodrigues. "On evaluating stream learning algorithms." Machine learning 90.3 (2013): 317-346.

CHAPTER

SIX

WALKTHROUGH 3: CREATING AN NGC CLASSIFIER

In this demonstration, we will learn how to create a classifier based on NGC. After going through this demonstration, you will:

1. Learn how to use a simple projection graph as well as the `extract()` and `inject()` routines to initialize the simulated settling process of an NGC model.
2. Craft and simulate an NGC model that can directly classify the image patterns in the MNIST database (from Demonstration # 1), producing results comparable to what was reported in (Whittington & Bogacz, 2017).

Note that the folders of interest to this demonstration are:

- `walkthroughs/demo3/`: this contains the necessary simulation script
- `walkthroughs/data/`: this contains the zipped copy of the MNIST database arrays

6.1 Using an Ancestral Projection Graph to Initialize the Settling Process

We will start by first discussing an important use-case of the `ProjectionGraph` – to initialize the simulated iterative inference process of an `NGCGraph`. This is contrast to the use-case we saw in the last two walkthroughs where we used the ancestral projection graph as a post-training tool, which allowed us to draw samples from the underlying directed generative models we were fitting. This time, we will leverage the power of an ancestral projection graph to serve as a simple, progressively improving model of initial conditions for an iterative inference process.

To illustrate the above use-case, we will focus on crafting an NGC model for discriminative learning (as opposed to the generative learning models we built Walkthroughs # 1 and #2). Before working with a concrete application, as we will do in the next section, let us just focus on crafting the NGC architecture of the classifier as well as its ancestral projection graph.

Working with nodes and cables (see *the last demonstration for details*), we will build a simple hierarchical system that adheres to the following NGC shorthand:

```
Node Name Structure:
z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
Note that z3 = x and z0 = y, which yields a classifier
```

where we see will design an NGC predictive processing model that contains three state layers `z0`, `z1`, and `z2` with the special application-specific usage that, during training, `z0` will be clamped to a label vector `y` (a one-hot encoding of a single category out of a finite set – `1-of-C` encoding, where `C` is the number of classes) and `z2` will be clamped to a sensory input vector `x`. Building the above NGC system entails writing the following:

```
batch_size = 128
x_dim = # dimensionality of input space
y_dim = # dimensionality of output/target space
beta = 0.1
leak = 0.0
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
# set up system nodes
z2 = SNode(name="z2", dim=x_dim, beta=beta, leak=leak, act_fx="identity",
           integrate_kernel=integrate_cfg)
mu1 = SNode(name="mu1", dim=z_dim, act_fx="identity", zeta=0.0)
e1 = ENode(name="e1", dim=z_dim)
z1 = SNode(name="z1", dim=z_dim, beta=beta, leak=leak, act_fx="relu6",
           integrate_kernel=integrate_cfg)
mu0 = SNode(name="mu0", dim=y_dim, act_fx="softmax", zeta=0.0)
e0 = ENode(name="e0", dim=y_dim)
z0 = SNode(name="z0", dim=y_dim, beta=beta, integrate_kernel=integrate_cfg, leak=0.0)

# create cable wiring scheme relating nodes to one another
wght_sd = 0.02
init_kernels = {"A_init" : ("gaussian",wght_sd), "b_init" : ("zeros")}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : 1234}
pos_scable_cfg = {"type": "simple", "coeff": 1.0} # a positive cable
neg_scable_cfg = {"type": "simple", "coeff": -1.0} # a negative cable

z2_mu1 = z2.wire_to(mu1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
mu1.wire_to(e1, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z1.wire_to(e1, src_comp="z", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e1.wire_to(z2, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z2_mu1,"A^T"))
e1.wire_to(z1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

z1_mu0 = z1.wire_to(mu0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
mu0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z1_mu0,"A^T"))
e0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

# set up update rules and make relevant edges aware of these
z2_mu1.set_update_rule(preact=(z2,"phi(z)"), postact=(e1,"phi(z)"), param=["A","b"])
z1_mu0.set_update_rule(preact=(z1,"phi(z)"), postact=(e0,"phi(z)"), param=["A","b"])

# Set up graph - execution cycle/order
model = NGCGraph(K=5)
model.set_cycle(nodes=[z2,z1,z0])
model.set_cycle(nodes=[mu1,mu0])
model.set_cycle(nodes=[e1,e0])
model.compile(batch_size=batch_size)
```

noting that `x_dim` and `y_dim` would be determined by your input dataset's sensory input design matrix `X` and its corresponding label design matrix `Y`. Also notice that, in our classifier above, because we will generally be clamping an input data vector (or batch of them) to `z2`, we chose to encode an `identity` activation function for that node (we do not want to arbitrarily apply a nonlinear transform to the input). The activation function of the output prediction node `mu0` (which will attempt to predict the value of data clamped at `z0`, i.e., the "label node") has been set to be the `softmax` which will induce a soft form of competition among the neurons in `mu0` and allow our NGC classifier to produce probability distribution vectors in its output.

The architecture above could then be readily simulated assuming that we always have an `x` and a `y` to clamp to its `z2` and `z0` nodes. While it is possible to then run the same system in the absence of a `y` (as in test-time inference), we would have to simulate the NGC system for a reasonable number of steps (which might be greater than the number of steps `K` chosen to facilitate learning) or until convergence to a fixed-point (or stable attractor). While this approach is fine in principle, it would be ideal for downstream application use if we could leverage the underlying directed generative model that the above architecture embodies. Specifically, even though we crafted our model with discriminative learning as our goal, the above system is still learning, "under the hood", a generative model, specifically a conditional generative model of the form `p(y|x)`. Given this insight, we can take advantage of the fact that ancestral sampling through our model is still possible, just with the exception that our input samples do not need to come from a prior distribution (as in the case of the models in Walkthroughs # 1 and # 2) but instead from data patterns directly.

To build the corresponding ancestral projection graph for the architecture above, we would then (adhering to our NGC shorthand and ensuring this co-model graph follows the information flow through our NGC system – a design principle/heuristic we discussed in Demonstration # 2) write the following:

```
# build this NGC model's sampling graph
z2_dim = ngc_model.getNode("z2").dim
z1_dim = ngc_model.getNode("z1").dim
z0_dim = ngc_model.getNode("z0").dim
# Set up complementary sampling graph to use in conjunction w/ NGC-graph
s2 = FNode(name="s2", dim=z2_dim, act_fx="identity")
s1 = FNode(name="s1", dim=z1_dim, act_fx=act_fx)
s0 = FNode(name="s0", dim=z0_dim, act_fx=out_fx)
s2_s1 = s2.wire_to(s1, src_comp="phi(z)", dest_comp="dz", mirror_path_kernel=(z2_mu1,"A
↪"))
s1_s0 = s1.wire_to(s0, src_comp="phi(z)", dest_comp="dz", mirror_path_kernel=(z1_mu0,"A
↪"))
sampler = ProjectionGraph()
sampler.set_cycle(nodes=[s2,s1,s0])
sampler.compile()
```

which explicitly instantiates the conditional generative embodied by the NGC system we built earlier, allowing us to easily sample from it. If one wanted NGC shorthand for the above conditional generative model, it would be:

```
Node Name Structure:
s2 -(s2-s1)-> s1 -(s1-s0-)-> s0
Note: s3 = x, which yields the model p(s0=y|x)
Note: s2-s1 = z2-mu1 and s1-s0 = z1-mu0
```

where we have highlighted that we are sharing (or shallow copying) the exact synaptic connections (`z2-mu1` and `z1-mu0`) from the NGC system above into those of our directed generative model (`s2-s1` and `s1-s0`). Note that, after training the earlier NGC system on a database of images and their respective labels, we could then classify, at test-time, each unseen pattern using the conditional generative model directly (instead of the settling process of the original NGC system), like so:

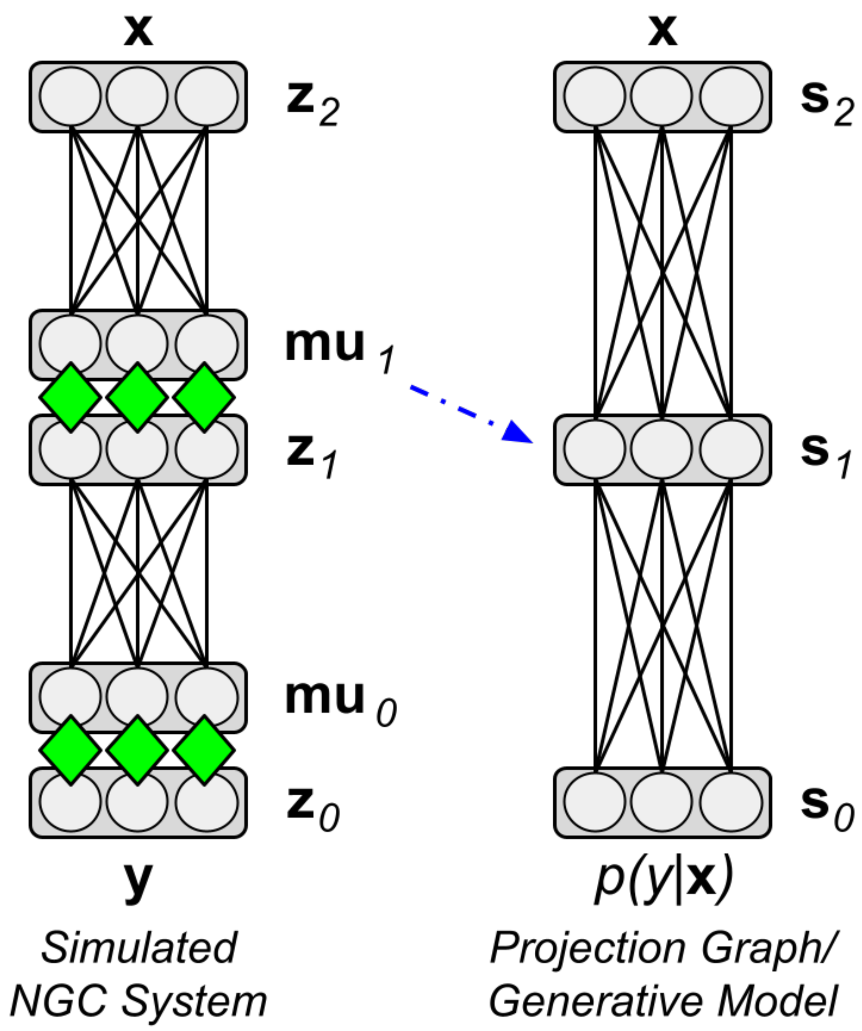
```
y = # test label/batch sampled from the test-set
x = # test data point/batch sampled from the test-set
readouts = sampler.project(
                clamped_vars=[("s2","z",x)],
                readout_vars=[("s0","phi(z)")]
            )
y_hat = readouts[0][2] # get probability distribution p(y|x)
```

Given our ancestral projection graph, we can now revisit our original goal of improving our NGC system's learning process by tying together it back with the simulation system itself. To do so, all we need to do is make use of two functions, i.e., `extract()` and `inject()`, provided by both the `ProjectionGraph` and `NGCGraph` objects. Tying together the two objects would then work as follows (below we emulate one step of online learning):

```
y = # test label/batch sampled from the test-set
x = # test data point/batch sampled from the test-set

# first, run the projection graph
readouts = sampler.project(
                clamped_vars=[("s2","z",x)],
                readout_vars=[("s0","phi(z)")]
            )

# second, extract data from the ancestral projection graph
s2 = sampler.extract("s2","z")
s1 = sampler.extract("s1","z")
s0 = sampler.extract("s0","z")

# third, initialize the simulated NGC system with the above information
model.inject([("mu1", "z", s1), ("z1", "z", s1), ("mu0", "z", s0)])

# finally, run/simulate the NGC system as normal
readouts, delta = model.settle(
                    clamped_vars=[("z2","z",x),("z0","z",y)],
                    readout_vars=[("mu0","phi(z)"),("mu1","phi(z)")]
                  )
y_hat = readouts[0][2]
for p in range(len(delta)):
    delta[p] = delta[p] * (1.0/(x.shape[0] * 1.0))
opt.apply_gradients(zip(delta, model.theta))
model.clear()
sampler.clear()
```

where we see that, after we first run the ancestral projection graph, we then extract the internal values from the `s1` and `s0` nodes (from their `z` compartments) and inject these into the relevant spots inside the NGC system, i.e., we place the reading in the `z` compartment of `s1` into the `z` compartment of `mu1` and `z1` (since we don't want the error neuron `e1` to find any mismatch in the first time step of the settling process of `model`) and the `z` compartment of `s0` into the `z` compartment of `mu0` (to ensure that, since we will be clamping `y` to the `z` compartment of `z0`, we want the mismatch signal simulated to be the different between bottom layer prediction `mu0` and the label at the very first time step of the settling process of `model`). In short, we just want the initial conditions of the settling process for `model` to be such that its state `z1` matches the expectation `mu1` of `z2` (clamped to `x`) and the expectation `mu0` of `z1` is being initially compared to the state of `z0` (clamped to the label `y`). Note that when values are "injected" into a NGC system through `inject()`, they will not persist after the first step of its settling process – they will evolve according to its current node dynamics. If you did not want a node to evolve at all remain fixed at the value you embed/insert, then you would use the `clamp()` function instead (which is what is being used internally to clamp variables in the `clamped_vars` argument of the `settle()` function above).

In the figure below, we graphically what the above simulated NGC system and its corresponding conditional generative model (ancestral projection graph) look like (the blue dashed arrow just point outs that the layer `s1` of the generative model is the same thing as the mean prediction `mu1` of the original NGC model).



The three-layer hierarchical classifier above turns out to be very similar to the one implemented in ngc-learn's Model Museum – the *GNCN-t1-FFM*, which is itself a four-layer discriminative NGC system that emulates the model investigated in [1]. We will import and use this slightly deeper model in the next part of this demonstration.

6.2 Learning a Classifier

Now that we have seen how to design an NGC classifier and build a projection graph that allows us to directly use the underlying conditional generative model of `p(y|x)`,
we have a powerful means to initialize our system's internal nodes to something meaningful and task-specific (instead of the default zero-vector initialization) as well as a fast label prediction model as an important by-product of the discriminative learning that our code will be doing. Having a fast model for test-time inference is useful not only for quickly tracking generalization ability throughout training (using a validation subset of data points) but also for downstream uses of the learning generative model – for example, one could extract the synaptic weight matrices inside the ancestral projection graph, serialize them to disk, and place them inside a multi-layer perceptron structure with the same layer sizes/architecture built pure Tensorflow or Pytorch.

Specifically, we will fit a supervised NGC classifier using the labels that come with the processed MNIST dataset, in `mnist.zip` (which you unzipped and worked with in Demonstration # 1). For this part of the demonstration, we will import the full model of [1], You will notice in the provided training script `sim_train.py`, we import the `GNCN-t1-FFM` (the NGC classifier model) in the header:

```
from ngclearn.museum.gncn_t1_ffm import GNCN_t1_FFM
```

which is initialized as later in the code as:

```
args = # ...the Config object loaded in earlier...

agent = GNCN_t1_FFM(args) # set up NGC model
```

and then proceed to write/design a training process very similar in design to the one we wrote in Demonstration # 1. The key notable differences are now that we are:

1. using labels along with the input sensory samples, meaning we need to tell the DataLoader management object that there is a label design matrix to sample that maps one-to-one with the image design matrix, like so:

```
xfname = # ...the design matrix X file name loaded in earlier...
yfname = # ...the design matrix Y file name loaded in earlier...
args = # ...the Config object loaded in earlier...
batch_size = # ... number of samples to draw from the loader per training step ...

# load data into memory
X = ( tf.cast(np.load(xfname),dtype=tf.float32) ).numpy()
x_dim = X.shape[1]
args.setArg("x_dim",x_dim) # set the config object "args" to know of the dimensionality␣
↪of x
Y = ( tf.cast(np.load(yfname),dtype=tf.float32) ).numpy()
y_dim = Y.shape[1]
args.setArg("y_dim",y_dim) # set the config object "args" to know of the dimensionality␣
↪of y
# build the training set data loader
train_set = DataLoader(design_matrices=[("z3",X),("z0",Y)], batch_size=batch_size)
```

2. we are now using the NGC model's ancestral projection graph to make label predictions in our eval_model() function and we now globally track Acc instead of ToD (since the projection graph does not have a total discrepancy quantity that we can measure) as well as Ly (the Categorical cross entropy of our model's label probabilities) instead of Lx. This is done (in eval_model()) as follows:

```
x = # ... image/batch drawn from data loader ...
y = # ... label/batch drawn from data loader ...

y_hat = agent.predict(x)

# update/track fixed-point losses
Ly = tf.reduce_sum( metric.cat_nll(y_hat, y) ) + Ly

# compute number of correct predictions in batch
y_ind = tf.cast(tf.argmax(y,1),dtype=tf.int32)
y_pred = tf.cast(tf.argmax(y_hat,1),dtype=tf.int32)
comp = tf.cast(tf.equal(y_pred,y_ind),dtype=tf.float32)
Acc += tf.reduce_sum( comp ) # update/track overall accuracy
```

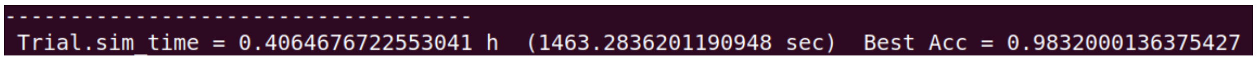
3. we finally tie together the ancestral projection graph with the NGC classifier's settling process during training. This is done through the code snippet below:

```
x = # ... image/batch drawn from data loader ...
y = # ... label/batch drawn from data loader ...

# run ancestral projection to get initial conditions
y_hat_ = agent.predict(x) # run p(y|x)
mu1 = agent.ngc_sampler.extract("s1","z") # extract value of s1
mu0 = agent.ngc_sampler.extract("s0","z") # extract value of s0

# set initial conditions for NGC system
agent.ngc_model.inject([("mu1", "z", mu1), ("z1", "z", mu1), ("mu0", "z", mu0)])

# conduct iterative inference/setting as normal
y_hat = agent.settle(x, y)
ToD_t = calc_ToD(agent) # calculate total discrepancy
Ly = tf.reduce_sum( metric.cat_nll(y_hat, y) ) + Ly

# update synaptic parameters given current model internal state
delta = agent.calc_updates()
opt.apply_gradients(zip(delta, agent.ngc_model.theta))
agent.ngc_model.apply_constraints()
agent.clear()
```

To train your NGC classifier, run the training script in `/walkthroughs/demo3/` as follows:

```
$ python sim_train.py --config=gncn_t1_ffm/fit.cfg --gpu_id=0 --n_trials=1
```

which will execute a training process using the experimental configuration file `/walkthroughs/demo3/gncn_t1_ffm/fit.cfg` written for you. After your model finishes training you should see a validation score similar to the one below:

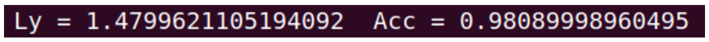


```
-----------------------------------
 Trial.sim_time = 0.4064676722553041 h  (1463.2836201190948 sec)  Best Acc = 0.9832000136375427
```

You will also notice that in your folder `/walkthroughs/demo3/gncn_t1_ffm/` several arrays as well as your learned NGC classifier have been saved to disk for you. To examine the classifier's performance on the MNIST test-set, you can execute the evaluation script like so:

```
$ python eval_model.py --config=gncn_t1_ffm/fit.cfg --gpu_id=0
```

which should result in an output similar to the one below:



```
Ly = 1.4799621105194092  Acc = 0.98089998960495
```

Desirably, our out-of-sample results on both the validation and test-set corroborate the measurements reported in (Whittington & Bogacz, 2017) [1], i.e., a range of `1.7-1.8`% validation error was reported and our simulation yields a validation accuracy of `0.9832 * 100 = 98.32`% (or `1.68`% error) and a test accuracy of `0.98099 * 100 = 98.0899`% (or about `1.91`% error), even though our predictive processing classifier/set-up differs in a few small ways:

1) we induce soft competition in the label prediction `mu0` with the `softmax` (whereas they used the `identity` function and softened the label vectors through clipping),

2) we work directly with the normalized pixel data whereas [1] transforms the data with an inverse logistic transform (you can find this function implemented as `inverse_logistic()` in `ngclearn.utils.transform_utils`), and

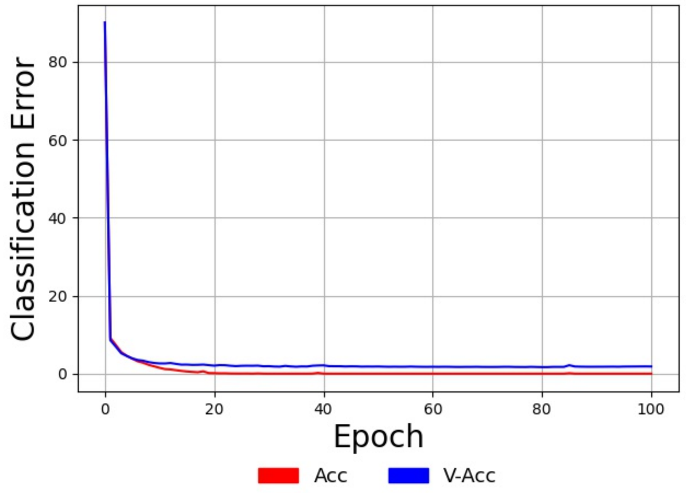
3) they initialize their weights using a scheme based on the Uniform distribution (or the `classic_glorot` scheme in `ngclearn.utils.transform_utils`). (Note that you can modify the scripts `sim_train.py`, `fit.cfg`,

and `eval_model.py` to incorporate these changes and obtain similar results under the same conditions.)

Finally, since we have collected our training and validation accuracy measurements at the end of each pass through the data (or epoch/iteration), we can run the following to obtain a plot of our model's learning curves:

```
$ python plot_curves.py
```

which, internally, has been hard-coded to point to the local directory `walkthroughs/demo3/gncn_t1_ffm/` containing the relevant measurements/numpy arrays. Doing so should result in a plot that looks similar to the one below:



As observed in the plot above, this NGC overfits the training sample perfectly (reaching a training error `0.0`%) as indicated by the fact that the blue validation `V-Acc` curve is a bit higher than the red `Acc` learning curve (which itself converges to and remains at perfect training accuracy). Note that these reported accuracy measurements come from the ancestral projection graph we used to initialize the settling process of the discriminative NGC system, meaning that we can readily deploy the projection graph itself as a direct probabilistic model of `p(y|x)`.

6.3 References

[1] Whittington, James CR, and Rafal Bogacz. "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity." Neural computation 29.5 (2017): 1229-1262.

CHAPTER

SEVEN

WALKTHROUGH 4: SPARSE CODING

In this demonstration, we will learn how to create, simulate, and visualize the internally acquired filters/atoms of variants of a sparse coding system based on the classical model proposed by (Olshausen & Field, 1996) [1]. After going through this demonstration, you will:

1. Learn how to build a 2-layer NGC sparse coding model of natural image patterns, using the original dataset used in [1].
2. Visualize the acquired filters of the learned dictionary models and examine the results of imposing a kurtotic prior as well as a thresholding function over latent codes.

Note that the folders of interest to this demonstration are:

- `walkthroughs/demo4/`: this contains the necessary simulation scripts
- `walkthroughs/data/`: this contains the zipped copy of the natural image arrays

7.1 On Dictionary Learning

Dictionary learning poses a very interesting question for statistical learning: can we extract "feature detectors" from a given database (or collection of patterns) such that only a few of these detectors play a role in reconstructing any given, original pattern/data point? The aim of dictionary learning is to acquire or learn a matrix, also called the "dictionary", which is meant to contain "atoms" or basic elements inside this dictionary (such as simple fundamental features such as the basic strokes/curves/edges that compose handwritten digits or characters). Several atoms (or rows of this matrix) inside the dictionary can then be linearly combined to reconstruct a given input signal or pattern. A sparse dictionary model is able to reconstruct input patterns with as few of these atoms as possible. Typical sparse dictionary or coding models work with an over-complete spanning set, or, in other words, a latent dimensionality (which one could think of as the number of neurons in a single latent state node of an NGC system) that is greater than the dimensionality of the input itself.

From a neurobiological standpoint, sparse coding emulates a fundamental property of neural populations – the activities among a neural population are sparse where, within a period of time, the number of total active neurons (those that are firing) is smaller than the total number of neurons in the population itself. When sensory inputs are encoded within this population, different subsets (which might overlap) of neurons activate to represent different inputs (one way to view this is that they "fight" or compete for the right to activate in response to different stimuli). Classically, it was shown in [1] that a sparse coding model trained on natural image patches learned within its dictionary non-orthogonal filters that resembled receptive fields of simple-cells (found in the visual cortex).

7.2 Constructing a Sparse Coding System

To build a sparse coding model, we can, as we have in the previous three walkthroughs, manually craft one using nodes and cables. First, let us specify the underlying generative model we aim to emulate. In NGC shorthand, this means that we seek to build:

```
Node Name Structure:
p(z1) ; z1 -(z1-mu0-)-> mu0 ;e0; z0
Note: Cauchy prior applied for p(z1)
```

Furthermore, we further specify underlying directed generative model (in accordance with the methodology in *Demonstration #3*) as follows:

```
Node Name Structure:
s1 -(s1-s0-)-> s0
Note: s1 ~ p(s1), where p(s1) is the prior over s1
Note: s1-s0 = z1-mu0
```

where we see that we aim to learn a two-layer generative system that specifically imposes a prior distribution `p(z1)` over the latent feature detectors that we hope to extract in node `z1`. Note that this two-layer model (or single latent-variable layer model) could either be the linear generative model from [1] or one similar to the model learned through ISTA [2] if a (soft) thresholding function is used instead.

Constructing the above system for (Olshausen & Field, 1996) is done, using nodes and cables, as follows:

```
x_dim = # ... dimension of patch data ...
# ---- build a sparse coding linear generative model with a Cauchy prior ----
K = 300
beta = 0.05
# general model configurations
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
prior_cfg = {"prior_type" : "cauchy", "lambda" : 0.14} # configure latent prior
# cable configurations
init_kernels = {"A_init" : ("unif_scale",1.0)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : seed}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}
neg_scable_cfg = {"type": "simple", "coeff": -1.0}
constraint_cfg = {"clip_type":"forced_norm_clip","clip_mag":1.0,"clip_axis":1}

# set up system nodes
z1 = SNode(name="z1", dim=100, beta=beta, leak=leak, act_fx=act_fx,
           integrate_kernel=integrate_cfg, prior_kernel=prior_cfg)
mu0 = SNode(name="mu0", dim=x_dim, act_fx=out_fx, zeta=0.0)
e0 = ENode(name="e0", dim=x_dim)
z0 = SNode(name="z0", dim=x_dim, beta=beta, integrate_kernel=integrate_cfg, leak=0.0)

# create the rest of the cable wiring scheme
z1_mu0 = z1.wire_to(mu0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z1_mu0.set_constraint(constraint_cfg)
mu0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z1_mu0,"symm_
↪tied"))
e0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)
```

(continues on next page)

(continued from previous page)

```
z1_mu0.set_update_rule(preact=(z1,"phi(z)"), postact=(e0,"phi(z)"), param=["A"])
param_axis = 1

# Set up graph - execution cycle/order
model = NGCGraph(K=K, name="gncn_t1_sc", batch_size=batch_size)
model.set_cycle(nodes=[z1,z0])
model.set_cycle(nodes=[mu0])
model.set_cycle(nodes=[e0])
model.compile()
```

while building its ancestral sampling co-model is done with the following code block:

```
# build this NGC model's sampling graph
z1_dim = ngc_model.getNode("z1").dim
z0_dim = ngc_model.getNode("z0").dim
s1 = FNode(name="s1", dim=z1_dim, act_fx=act_fx)
s0 = FNode(name="s0", dim=z0_dim, act_fx=out_fx)
s1_s0 = s1.wire_to(s0, src_comp="phi(z)", dest_comp="dz", mirror_path_kernel=(z1_mu0,
↪"tied"))
sampler = ProjectionGraph()
sampler.set_cycle(nodes=[s1,s0])
sampler.compile()
```

Notice that we have, in our `NGCGraph`, taken care to set the `.param_axis` variable to be equal to 1 – this will, whenever we call `apply_constraints()`, tell the NGC system to normalize the Euclidean norm of the columns of each generative/forward matrix to be equal to `.proj_weight_mag` (which we set to the typical value of 1). This is a particularly important constraint to apply to sparse coding models as this prevents the trivial solution of simply growing out the magnitude of the dictionary synapses to solve the underlying constrained optimization problem (and, in general, constraining the rows or columns of NGC generative models helps to facilitate a more stable training process).

To build the version of our model using a thresholding function (instead of using a factorial prior over the latents), we can write the following:

```
x_dim = # ... dimension of image data ...
K = 300
beta = 0.05
# general model configurations
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
# configure latent threshold function
thr_cfg = {"threshold_type" : "soft_threshold", "thr_lambda" : 5e-3}
# cable configurations
dcable_cfg = {"type": "dense", "init" : ("unif_scale",1.0), "seed" : seed}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}
neg_scable_cfg = {"type": "simple", "coeff": -1.0}
constraint_cfg = {"clip_type":"forced_norm_clip","clip_mag":1.0,"clip_axis":1}

# set up system nodes
z1 = SNode(name="z1", dim=100, beta=beta, leak=leak, act_fx=act_fx,
           integrate_kernel=integrate_cfg, threshold_kernel=thr_cfg)
mu0 = SNode(name="mu0", dim=x_dim, act_fx=out_fx, zeta=0.0)
e0 = ENode(name="e0", dim=x_dim)
z0 = SNode(name="z0", dim=x_dim, beta=beta, integrate_kernel=integrate_cfg, leak=0.0)
```

(continues on next page)

(continued from previous page)

```
# create the rest of the cable wiring scheme
z1_mu0 = z1.wire_to(mu0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z1_mu0.set_constraint(constraint_cfg)
mu0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z1_mu0,"symm_
↪tied"))
e0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)
z1_mu0.set_update_rule(preact=(z1,"phi(z)"), postact=(e0,"phi(z)"), param=["A"])

# Set up graph - execution cycle/order
model = NGCGraph(K=K, name="gncn_t1_sc", batch_size=batch_size)
model.set_cycle(nodes=[z1,z0])
model.set_cycle(nodes=[mu0])
model.set_cycle(nodes=[e0])
model.compile()
```

Note that the ancestral projection this model using thresholding would be the same as the one we built earlier. Notably, the above models can also be imported from the Model Museum, specifically using *GNCN-t1/SC*, which internally implements the `NGCGraph`(s) depicted above.

Finally, for both the first model (which emulates [1]) and the second model (which emulates [2]), we should define their total discrepancy (ToD) measurement functions so we can track their performance throughout simulation:

```
def calc_ToD(agent, lmda):
    """Measures the total discrepancy (ToD), or negative energy, of an NGC system"""
    z1 = agent.ngc_model.extract(node_name="z1", node_var_name="z")
    e0 = agent.ngc_model.extract(node_name="e0", node_var_name="phi(z)")
    z1_sparsity = tf.reduce_sum(tf.math.abs(z1)) * lmda # sparsity penalty term
    L0 = tf.reduce_sum(tf.math.square(e0)) # reconstruction term
    ToD = -(L0 + z1_sparsity)
    return float(ToD)
```

In fact, the above total discrepancy, in the case of a sparse coding model, measures the negative of its underlying energy function, which is simply the sum of its reconstruction error (or the sum of the square of the NGC system's sensory error neurons `e0`) and the sparsity of its single latent state layer `z1`.

7.3 Learning Latent Feature Detectors

We will now simulate the learning of the feature detectors using the two sparse coding models that we have built above. The code provided in `sim_train.py` in `/walkthroughs/demo4/` will execute a simulation of the above two models on the natural images found in `walkthroughs/data/natural_scenes.zip`), which is a dataset composed of several images of the American Northwest.

First, navigate to the `walkthroughs/` directory to access the example/demonstration code and further enter the `walkthroughs/data/` sub-folder. Unzip the file `natural_scenes.zip` to create one more sub-folder that contains two numpy arrays, the first labeled `natural_scenes/raw_dataX.npy` and another labeled as `natural_scenes/dataX.npy`. The first one contains the original, `512 x 512` raw pixel image arrays (flattened) while the second contains the pre-processed, whitened/normalized (and flattened) image data arrays (these are the pre-processed image patterns used in [1]). You will, in this demonstration, only be working with `natural_scenes/dataX.npy`. Two (raw) images sampled from the original dataset (`raw_dataX.npy`) are shown below:

With the data unpacked and ready, we can now turn our attention to simulating the training process. One way to write the training loop for our sparse coding models would be the following:

```python
args = # load in Config object with user-defined arguments
args.setArg("batch_size",num_patches)
agent = GNCN_t1_SC(args) # set up NGC model
opt = tf.keras.optimizers.SGD(0.01) # set up optimization process

############################################################################
# create a  training loop
ToD, Lx = eval_model(agent, train_set, calc_ToD, verbose=True)
vToD, vLx = eval_model(agent, dev_set, calc_ToD, verbose=True)
print("{} | ToD = {}  Lx = {} ; vToD = {}  vLx = {}".format(-1, ToD, Lx, vToD, vLx))

########################################################################
mark = 0.0
for i in range(num_iter): # for each training iteration/epoch
    ToD = Lx = 0.0
    n_s = 0
    # run single epoch/pass/iteration through dataset
    ####################################################################
    for batch in train_set:
        x_name, x = batch[0]
        # generate patches on-the-fly for sample x
        x_p = generate_patch_set(x, patch_size, num_patches)
        x = x_p
        n_s += x.shape[0] # track num samples seen so far
        mark += 1

        x_hat = agent.settle(x) # conduct iterative inference
        ToD_t = calc_ToD(agent, lmda) # calc ToD
```

(continues on next page)

(continued from previous page)

```
        ToD = ToD_t + ToD
        Lx = tf.reduce_sum( metric.mse(x_hat, x) ) + Lx

        # update synaptic parameters given current model internal state
        delta = agent.calc_updates(avg_update=False)
        opt.apply_gradients(zip(delta, agent.ngc_model.theta))
        agent.ngc_model.apply_constraints()
        agent.clear()

        print("\r train.ToD {}  Lx {}  with {} samples seen (t = {})".format(
              (ToD/(n_s * 1.0)), (Lx/(n_s * 1.0)), n_s, (inf_time/mark)),
              end=""
              )
    ####################################################################
    print()
    ToD = ToD / (n_s * 1.0)
    Lx = Lx / (n_s * 1.0)
    # evaluate generalization ability on dev set
    vToD, vLx = eval_model(agent, dev_set, calc_ToD)
    print("-------------------------------------------------")
    print("{} | ToD = {}  Lx = {} ; vToD = {}  vLx = {}".format(
          i, ToD, Lx, vToD, vLx)
          )
```

notice that the training code above, which has also been integrated into the provided `sim_train.py` demo file, looks very similar to how we trained our generative models in *Demonstration # 1*. In contrast to our earlier training loops, however, we have now written and used patch creation function `generate_patch_set()` to sample image patches of `16 x 16` pixels on-the-fly each time an image is sampled from the `DataLoader`. Note that we have hard-coded this patch-shape, as well as the training `batch_size = 1` (since mini-batches of data are supposed to contain multiple patches instead of images), into `sim_train.py` in order to match the setting of [1]. As a result, the sparse coding training process consists of the following steps:

1) sample a random image from the image design matrix inside of the `DataLoader`,
2) generate a number of patches equal to `num_patches = 250` (which we have also hard-coded into `sim_train.py`), and
3) feed this mini-batch of image patches to the NGC system to facilitate a learning step.

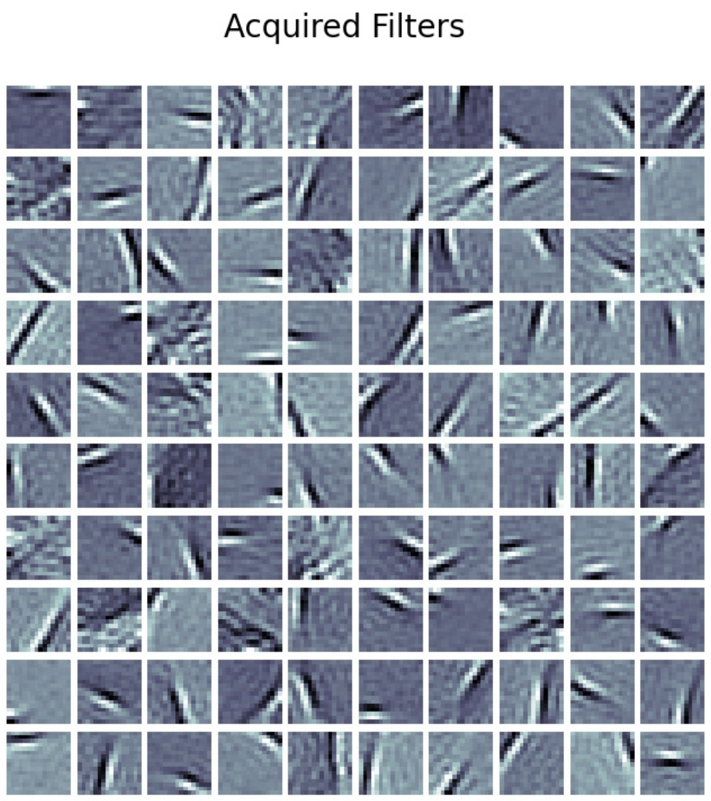
To train the first sparse coding model with the Cauchy factorial prior over `z1`, run the following the script:

```
$ python sim_train.py --config=sc_cauchy/fit.cfg --gpu_id=0 --n_trials=1
```

which will train a GNCN-t1/SC (with a Cauchy prior) on `16 x 16` pixel patches from the natural image dataset in [1]. After the simulation terminates, i.e., once `400` iterations/passes through the data have been made, you will notice in the `sc_cauchy/` sub-directory you have several useful files. Among these files, what we want is the serialized, trained sparse coding model `model0.ngc`. To extract and visualize the learned filters of this NGC model, you then need to run the final script, `viz_filters.py`, as follows:

```
$ python viz_filters.py --model_fname=sc_cauchy/model0.ngc --output_dir=sc_cauchy/
```

which will iterate through your model's dictionary atoms (stored within its single synaptic weight matrix) and ultimately produce a visual plot of the filters which should look like the one below:



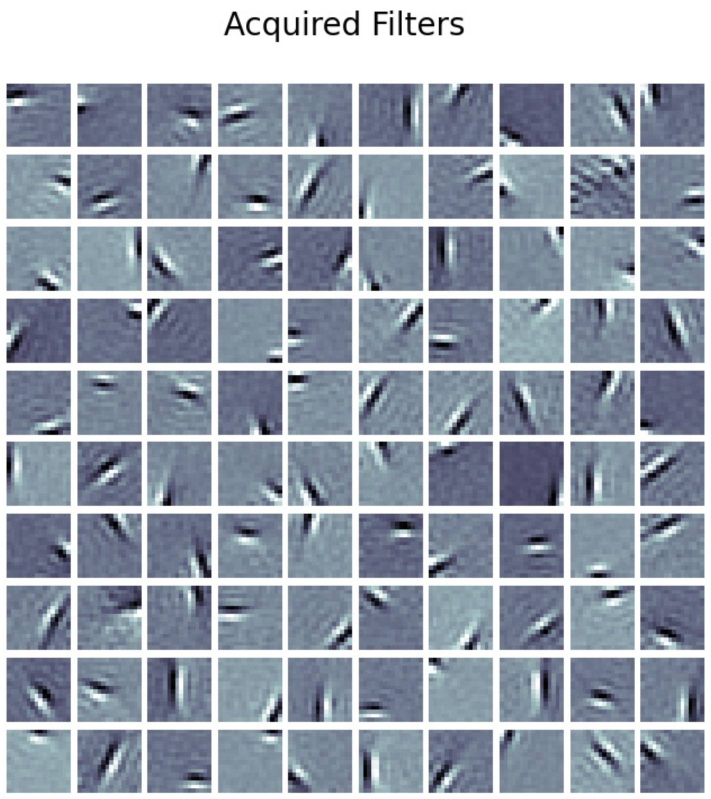
Now re-run the simulation but use the `sc_ista/fit.cfg` configuration instead, like so:

```
$ python sim_train.py --config=sc_ista/fit.cfg --gpu_id=0 --n_trials=1
```

and this will train your sparse coding using a latent soft-thresholding function (emulating ISTA). After this simulated training process ends, again, like before, run:

```
$ python viz_filters.py --model_fname=sc_ista/model0.ngc --output_dir=sc_ista/
```

and you should obtain a filter plot like the one below:



The filter plots, notably, visually indicate that the dictionary atoms in both sparse coding systems learned to function as edge detectors, each tuned to a particular position, orientation, and frequency. These learned feature detectors, as discussed in [1], appear to behave similar to the primary visual area (V1) neurons of the cerebral cortex in the brain. Although, in the end, the edge detectors learned by both our models qualitatively appear to be similar, we should note that the latent codes (when inferring them given sensory input) for the model that used the thresholding function are ultimately sparser. Furthermore, the filters for the model with thresholding appear to smoother and with fewer occurrences of less-than-useful slots than the Cauchy model (or filters that did not appear to extract any particularly interpretable features).

This difference in sparsity can be verified by examining the difference/gap between the absolute value of the total discrepancy `ToD` and the reconstruction loss `Lx` (which would tell us the degree of sparsity in each model since, according to our energy function formulation earlier, `|ToD| = Lx + lambda * sparsity_penalty`). In the experiment we ran for this demonstration, we saw that for the Cauchy prior model, at the start of training, the `|ToD|` was `14.18` and `Lx` was `12.42` (in nats) and, at the end of training, the `|ToD|` was `5.24` and `Lx` was `2.13` with the ending gap being `|ToD| - Lx = 3.11` nats. With respect to the latent thresholding model, we observed that, at the start, `|ToD|` was `-12.82` and `Lx` was `12.77` and, at the end, the `|ToD|` was `2.59` and `Lx` was `2.50` with the ending gap being `|ToD| - Lx = 0.09` nats. The final gap of the thresholding model is substantially lower than the one of the Cauchy prior model, indicating that the latent states of the thresholding model are, indeed, the sparsest.

7.4 References

[1] Olshausen, B., Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996). [2] Daubechies, Ingrid, Michel Defrise, and Christine De Mol. "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint." Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences 57.11 (2004): 1413-1457.

CHAPTER

EIGHT

WALKTHROUGH 5: AMORTIZED INFERENCE

In this demonstration, we will design a simple way to conduct amortized inference to speed up the settling process of an NGC model, cutting down the number of steps needed overall. We will build a custom model, which we will call the hierarchical ISTA model or "GNCN-t1-ISTA", and train it on the Olivetti database of face images [4]. After going through this demonstration, you will:

1. Learn how to construct a learnable inference projection graph to initialize the states of an NGC system, facilitating amortized inference.
2. Design a deep sparse coding model for modeling faces using the original dataset used in [4] and visualize the acquired filters of the learned representation system.

Note that the folders of interest to this demonstration are:

- `walkthroughs/demo5/`: this contains the necessary simulation scripts
- `walkthroughs/data/`: this contains the zipped copy of the face image arrays

8.1 Speeding Up the Settling Process with Amortized Inference

Although fitting an NGC model (a GNCN) to a data sample is a rather straightforward process, as we saw in the Demo 1 the underlying dynamics of the neural system require performing K steps of an iterative settling (inference) process to find suitable estimates of the latent neural state values. For the problem we have investigated so far, this only required around 50 steps which is not too expensive to simulate but for higher-dimensional, more complex problems, such as modeling temporal data generating processes or learning from sparse signals (as in the case of reinforcement learning), this settling process could potentially start maxing out modest computational budgets.

There are, at least, two key paths to reduce the underlying computational expense of the iterative settling process required by a predictive processing NGC model:

1) exploit the layer-wise parallelism inherent to the NGC state and synaptic update calculations – since NGC models are not update-locked (the state predictions and weight updates do not depend on one another) as deep neural networks are, one could design a distributed algorithm where a group/system of GPUs/CPUs synchronously (or asynchronously) compute(s) layer-wise predictions and weight updates, and
2) reduce the number of settling steps by constructing a computation process that infers the values of the latent states of a GNCN given a sensory sample(s), ultimately serving as an intelligent initialization of the state values instead of starting from zero vectors. For this second way, approaches have ranged from ancestral sampling/projection, as in deep Boltzmann machines [1] and as in for NGC systems formulated for active inference [2], to learning (jointly with the generative model) a complementary (neural) model, sometimes called a "recognition model", in a process known as amortized inference, e.g., in sparse coding the algorithm developed to do this was called predictive sparse decomposition [3]. Amortize means, in essence, to gradually reduce the initial cost of something (whether it be an asset or activity) over a period.

While there are many ways in which one could implement amortized inference, we will focus on using ngc-learn's ProjectionGraph to construct a simple, learnable recognition model.

8.2 The Model: Hierarchical ISTA

We will start by first constructing the model we would like to learn. Specifically, for this demonstration, we want to build a model for synthesizing human faces, specifically those contained in the Olivetti faces database.

For this part of the demonstration, you will need to unzip the data contained in walkthroughs/data/faces.zip (in the walkthroughs/data/ sub-folder) to create the necessary sub-folder which contains a single numpy array, faces/dataX.npy. This data file contains the flattened vectors of 40 images of size 256 x 256 pixels (pixel values have been normalized to the range of [0,1]), each depicting a human face. Two images sampled from the dataset (dataX.npy) are shown below:

We will now construct the specialized model which we will call, in the context of this demonstration, the "GNCN-t1-ISTA" (or "deep ISTA"). Specifically, we will extend our sparse coding ISTA model from Demonstration #4 to utilize an extra layer of latent variables "above". Notably, we will use the soft-thresholding function, which can be viewed as inducing a form a local lateral competition in the latent activities to yield sparse representations, and apply to the two latent state nodes of our system.

We start by first specifying the NGC system in design shorthand:

```
Node Name Structure:
z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
```

where we see that our three-layer system consists of seven nodes in total, i.e., the three latent state nodes z2, z1 and z0, the two mean prediction nodes mu1 and mu0, and the two error neuron nodes e1 and e0. Note that, when we build our recognition model later, our goal will be to infer good guess of the initial values of the z compartment of the nodes z1 and z2 (with z0 being clamped to the input image patch x).

Inside of the provided gncn_t1_ista.py, we see how the core of the system was put together with nodes and cables to create the hierarchical generative model:

```
x_dim = # ... dimension of patch data ...
# ---- build a hierarchical ISTA model ----
K = 10
beta = 0.05
# general model configurations
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : True}
thr_cfg = {"threshold_type" : "soft_threshold", "thr_lambda" : 5e-3}
```

(continues on next page)

(continued from previous page)

```
# cable configurations
init_kernels = {"A_init" : ("unif_scale",1.0)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : seed}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}
neg_scable_cfg = {"type": "simple", "coeff": -1.0}
constraint_cfg = {"clip_type":"forced_norm_clip","clip_mag":1.0,"clip_axis":1}

# set up system nodes
z2 = SNode(name="z2", dim=100, beta=beta, leak=0, act_fx="identity",
           integrate_kernel=integrate_cfg, threshold_kernel=thr_cfg)
mu1 = SNode(name="mu1", dim=100, act_fx="identity", zeta=0.0)
e1 = ENode(name="e1", dim=100)
z1 = SNode(name="z1", dim=100, beta=beta, leak=0, act_fx="identity",
           integrate_kernel=integrate_cfg, threshold_kernel=thr_cfg)
mu0 = SNode(name="mu0", dim=x_dim, act_fx="identity", zeta=0.0)
e0 = ENode(name="e0", dim=x_dim)
z0 = SNode(name="z0", dim=x_dim, beta=beta, integrate_kernel=integrate_cfg, leak=0.0)

# set up latent layer 2 to layer 1
z2_mu1 = z2.wire_to(mu1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z2_mu1.set_constraint(constraint_cfg)
mu1.wire_to(e1, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z1.wire_to(e1, src_comp="z", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e1.wire_to(z2, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z2_mu1,"A^T"))
e1.wire_to(z1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

# set up latent layer 1 to layer 0
z1_mu0 = z1.wire_to(mu0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=dcable_cfg)
z1_mu0.set_constraint(constraint_cfg)
mu0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z1_mu0,"A^T"))
e0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)

# set up update rules and make relevant edges aware of these
z2_mu1.set_update_rule(preact=(z2,"phi(z)"), postact=(e1,"phi(z)"), param=["A"])
z1_mu0.set_update_rule(preact=(z1,"phi(z)"), postact=(e0,"phi(z)"), param=["A"])

# Set up graph - execution cycle/order
print(" > Constructing NGC graph")
model = NGCGraph(K=K, name="gncn_t1_ista")
model.set_cycle(nodes=[z2,z1,z0])
model.set_cycle(nodes=[mu1,mu0])
model.set_cycle(nodes=[e1,e0])
model.apply_constraints()
model.compile(batch_size=batch_size)
```

Notice that we have set the number of simulated settling steps `K` to be quite small compared to the sparse coding models in Demonstration #4, i.e., we have drastically cut down the number of inference steps we required from `K = 300` to `K = 10`, a highly desirable `96.6`% decrease in computational cost (with respect to number of settling steps). The key is that the recognition model will learn to approximate the end-result of the settling process, and, over the course of training to an image database, progressively improve its estimates which will in turn better initialize the `NGCGraph` object's iterative inference. Since the recognition model will continually chase the result of the ever-improving settling process, we short-circuit the need for longer simulated settling processes with the trade-off that our iterative inference will be a bit less accurate in general (if the recognition model, which starts off randomly initialized, provides bad starting points in the latent search space, then the settling process will have to work harder to correct for the recognition model's deficiencies).

8.3 Constructing the Recognition Model

Building a recognition model for an NGC system is straightforward if we simply treat it as an ancestral projection graph with the key exception that it is "learnable". Specifically, we will randomly initialize an ancestral projection graph that will compute initial "guesses" of the activity values of `z1` and `z2` in our deep ISTA model. It helps to, as we did with the generative model, specify the form of the recognition model in shorthand as follows:

```
Node Name Structure:
s0 -s0-s1-> s1 ; s1 -s1-s2-> s2
Note: s1; e1_i ; z1, s2; e2_i ; z2
Note: s0 = x  // (we clamp s0 to data)
```

where we emphasize the difference between the recognition model and the generative model by labeling the recognition model's first and second latent layers as `s1` and `s2`, respectively. Our recognition model's goal, as explained before, will be to make its predicted value for `s1` match `z1` as well as make its predicted value `s2` match `z2`, where `z1` and `z2` are the results of the `NGCGraph` model's setting process that we designed above. This matching task is emphasized by our shorthand's second line, where we see that the value of `s1` will be compared to `z1` via the error node `e1_i` and `s2` will be compared to `z2` via `e2_i`.

Unlike the previous projection graphs we have built in earlier walkthroughs, our recognition model runs in the "opposite" direction of our generative model – it takes in data and predicts initial values for the latent states while the generative model predicts a value for the data given the latent states. Together, the recognition and the generative model will learn to cooperate in order to produce reasonable values for the latent states `z1` and `z2` that could plausibly produce a given input image patch `z0 = x`.

To create the recognition model that will allow us to conduct amortized inference, we write the following:

```
# set up this NGC model's recognition model
inf_constraint_cfg = {"clip_type":"norm_clip","clip_mag":1.0,"clip_axis":0}
z2_dim = ngc_model.getNode("z2").dim
z1_dim = ngc_model.getNode("z1").dim
z0_dim = ngc_model.getNode("z0").dim

s0 = FNode(name="s0", dim=z0_dim, act_fx="identity")
s1 = FNode(name="s1", dim=z1_dim, act_fx="identity")
st1 = FNode(name="st1", dim=z1_dim, act_fx="identity")
s2 = FNode(name="s2", dim=z2_dim, act_fx="identity")
st2 = FNode(name="st2", dim=z2_dim, act_fx="identity")
s0_s1 = s0.wire_to(s1, src_comp="phi(z)", dest_comp="dz", cable_kernel=dcable_cfg)
s0_s1.set_constraint(inf_constraint_cfg)
s1_s2 = s1.wire_to(s2, src_comp="phi(z)", dest_comp="dz", cable_kernel=dcable_cfg)
s1_s2.set_constraint(inf_constraint_cfg)

# build the error neurons that examine how far off the inference model was
# from the final NGC system's latent activities
e1_inf = ENode(name="e1_inf", dim=z_dim)
s1.wire_to(e1_inf, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
st1.wire_to(e1_inf, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_
↪cfg)
```

(continues on next page)

(continued from previous page)

```
e2_inf = ENode(name="e2_inf", dim=z_dim)
s2.wire_to(e2_inf, src_comp="phi(z)", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
st2.wire_to(e2_inf, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_
↪cfg)

# set up update rules and make relevant edges aware of these
s0_s1.set_update_rule(preact=(s0,"phi(z)"), postact=(e1_inf,"phi(z)"), param=["A"])
s1_s2.set_update_rule(preact=(s1,"phi(z)"), postact=(e2_inf,"phi(z)"), param=["A"])

sampler = ProjectionGraph()
sampler.set_cycle(nodes=[s0,s1,s2])
sampler.set_cycle(nodes=[st1,st2])
sampler.set_cycle(nodes=[e1_inf,e2_inf])
sampler.compile()
```

Now all that remains is to combine the recognition model with the generative model to create the full system. Specifically, to tie the two components together, we would write the following code:

```
x = # ... sampled image patch (or batch of patches) ...
# run recognition model
readouts = sampler.project(
                clamped_vars=[("s0","z",x)],
                readout_vars=[("s1","z"),("s2","z")]
            )
s1 = readouts[0][2]
s2 = readouts[1][2]
# now run the settling process
readouts, delta = model.settle(
                    clamped_vars=[("z0","z", x)],
                    init_vars=[("z1","z",s1),("z2","z",s2)],
                    readout_vars=[("mu0","phi(z)"),("z1","z"),
                                  ("z2","z")],
                    calc_delta=True
                  )
x_hat = readouts[0][2]

# now compute the updates to the encoder given the current state of system
z1 = readouts[1][2]
z2 = readouts[2][2]
#z3 = readouts[3][2]
sampler.project(
    clamped_vars=[("s0","z",tf.cast(x,dtype=tf.float32)),
                  ("s1","z",s1),("s2","z",s2),
                  ("st1","z",z1),("st2","z",z2)]
)
r_delta = sampler.calc_updates()

# update NGC system synaptic parameters
opt.apply_gradients(zip(delta, model.theta))
# update recognition model synaptic parameters
r_opt.apply_gradients(zip(delta, sampler.theta))
```

The above code snippet would generally occur within your training loop (which would be the same as the one in Demonstration #4) and can be founded integrated into the two key files provided for this demonstration, i.e., `sim_train.py` and `gncn_t1_ista.py`. Note that the `gncn_t1_ista.py` further illustrates how you can write a model that would fit within the general schema of ngc-learn's Model Museum, which requires that NGC systems provide an API to their key task-specific functions. `gncn_t1_ista.py` specifically implements all of the code we developed above for the deep ISTA model and its corresponding recognition model while `sim_train.py` is used to fit the model to the Olivetti dataset you unzipped into the `walkthroughs/data/` directory.

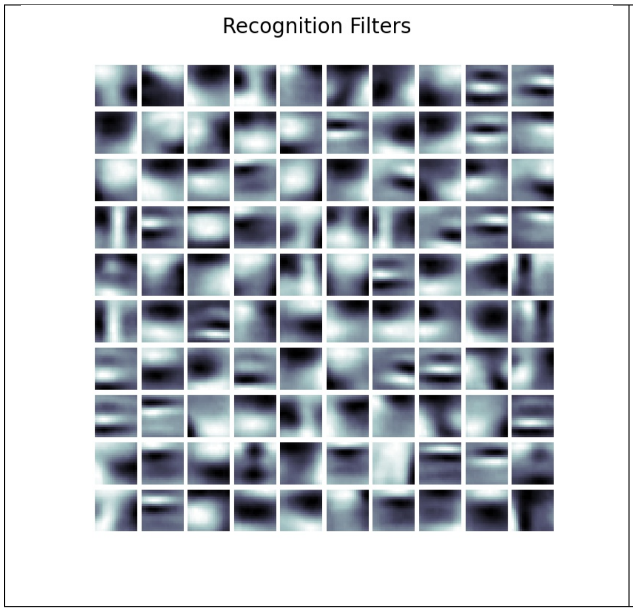
To train our deep ISTA model, you should execute the following:

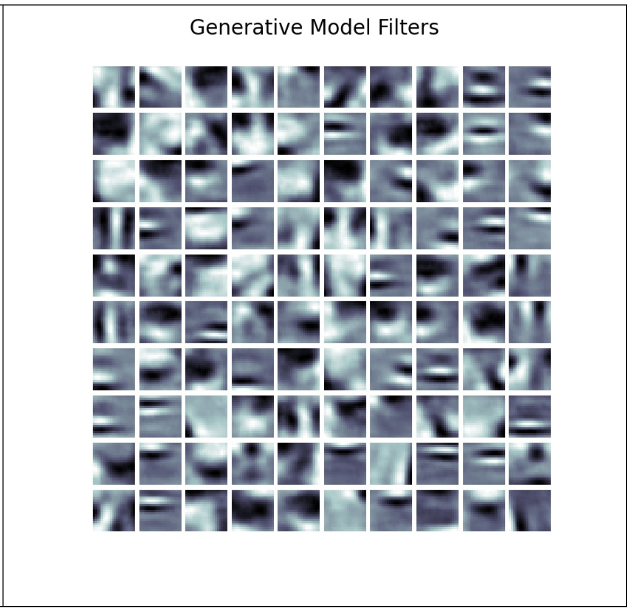
```
python sim_train.py --config=sc_face/fit.cfg --gpu_id=0
```

which will simulate the training of a deep ISTA model on face image patches for about `20` iterations. After this simulated process ends, you can then run the visualization script we have created for you:

```
$ python viz_filters.py --model_fname=sc_face/model0.ngc --output_dir=sc_face/ --viz_
↪encoder=True
```

which will produce and save two visualizations in your `sc_face/` sub-directory, one plot that depicts the learned bottom layer filters for the recognition model and one for the deep ISTA model. You should see filter plots similar to those presented below:





As we see, our NGC system has desirably learned low-level feature detectors corresponding to "pieces" of human faces, such as lips, noses, eyes, and other facial components. This was all learned only a few steps of simulated settling (`K = 10`) utilizing our learned recognition model. Notice that the low-level filters of the recognition model (the plot to the left) look similar to those acquired by the generative model but are "simpler" or less distinguished/sharp. This makes sense given that we designed our recognition model to "serve" the generative model by providing an initialization of its latent states (or "starting points" for the search for good latent states that generate the input patches). It appears that the recognition model's facial feature detectors are broad or less-detailed versions of those contained within our hierarchical ISTA model.

8.4 References

[1] Srivastava, Nitish, Ruslan Salakhutdinov, and Geoffrey Hinton. “Modeling documents with a Deep Boltzmann Machine.” Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (2013). [2] Ororbia, A. G. & Mali, A. Backprop-free reinforcement learning with active neural generative coding. In Proceedings of the AAAI Conference on Artificial Intelligence Vol. 36 (2022). [3] Kavukcuoglu, Koray, Marc’Aurelio Ranzato, and Yann LeCun. “Fast inference in sparse coding algorithms with applications to object recognition.” arXiv preprint arXiv:1010.3467 (2010). [4] Samaria, Ferdinando S., and Andy C. Harter. “Parameterisation of a stochastic model for human face identification.” Proceedings of 1994 IEEE workshop on applications of computer vision (1994).

CHAPTER

NINE

WALKTHROUGH 6: HARMONIUMS AND CONTRASTIVE DIVERGENCE

Although ngc-learn was originally designed with a focus on predictive processing neural systems, it is possible to simulate other kinds of neural systems with different dynamics and forms of learning. Notably, a class of learning and inference systems that adapt through a process known as contrastive Hebbian learning (CHL) can be constructed and simulated with ngc-learn.

In this walkthrough, we will design a simple (single-wing) Harmonium, also known as the restricted Boltzmann machine (RBM). We will specifically focus on learning its synaptic connections with an algorithmic recipe known as Contrastive Divergence (CD). After going through this walkthrough, you will:

1. Learn how to construct an `NGCGraph` that emulates the structure of an RBM and adapt the NGC settling process to calculate approximate synaptic weight gradients in accordance to Contrastive Divergence.
2. Simulate fantasized image samples using the block Gibbs sampler implicitly defined by the negative phase graph.

Note that the folders of interest to this walkthrough are:

- `walkthroughs/demo6/`: this contains the necessary simulation scripts
- `walkthroughs/data/`: this contains the zipped copy of the digit image arrays

9.1 On Single-Wing Harmoniums

A Harmonium is a generative model implemented as a stochastic, two-layer neural system that attempts to learn a probability distribution over sensory input $\mathbf{x}$, i.e., the goal of a Harmonium is to learn $p(\mathbf{x})$, much like the models we were learning in Walkthrough #1. Fundamentally, the approach to estimating $p(\mathbf{x})$ that is taken by a Harmonium is through optimizing an energy function $E(\mathbf{x})$ (a concept motivated by statistical mechanics), where the system searches for an internal configuration, i.e., the values of its synapses, has low energy (values) for patterns that lie come from the true data distribution $p(\mathbf{x})$ and high energy (values) for patterns that do not (or those that do not come from the training dataset).

The most common, standard Harmonium is one where input nodes (one per dimension of the data observation space) are modeled as binary/Boolean sensors, or "visible units" $\mathbf{z}^0$ (which are clamped to actual data patterns), connected to a layer of (stochastic) binary latent feature detectors, or "hidden units" $\mathbf{z}^1$. Notably, the connections between the latent and visible units are symmetric. As a result of a key restriction imposed on the Harmonium's network structure, i.e., no lateral connections between the neurons in $\mathbf{z}^0$ as well as those in $\mathbf{z}^1$, computing the latent and visible states is simple:

$$p(\mathbf{z}^1|\mathbf{z}^0) = sigmoid(\mathbf{W} \cdot \mathbf{z}^0 + \mathbf{c}), \ \mathbf{z}^1 \sim p(\mathbf{z}^1|\mathbf{z}^0)$$
$$p(\mathbf{z}^0|\mathbf{z}^1) = sigmoid(\mathbf{W}^T \cdot \mathbf{z}^1 + \mathbf{b}), \ \mathbf{z}^0 \sim p(\mathbf{z}^0|\mathbf{z}^1)$$

where $\mathbf{b}$ is the visible bias vector, $\mathbf{c}$ is the latent bias vector, and $\mathbf{W}$ is the synaptic weight matrix that connects $\mathbf{z}^0$ to $\mathbf{z}^1$ (and its transpose $\mathbf{W}^T$ is used to make predictions of the input itself). Note that $\cdot$ means matrix/vector multiplication and $\sim$ denotes that we would sample from a probability (vector) and, in the above Harmonium's case, samples will be drawn treating conditionals such as $p(\mathbf{z}^1|\mathbf{z}^0)$ as multivariate Bernoulli distributions. $\mathbf{z}^0$ would typically be clamped/set to the actual sensory input data $\mathbf{x}$.

The energy function of the Harmonium's joint configuration $(\mathbf{z}^0, \mathbf{z}^1)$ (similar to that of a Hopfield network) is specified as follows:

$$E(\mathbf{z}^0, \mathbf{z}^1) = -\sum_i \mathbf{b}_i \mathbf{z}^0_i - \sum_j \mathbf{c}_j \mathbf{z}^1_j - \sum_i \sum_j \mathbf{z}^0_i \mathbf{W}_{ij} \mathbf{z}^1_j$$

Notice in the equation above, we sum over indices, e.g., $\mathbf{z}^0_i$ retrieves the ith scalar element of (vector) $\mathbf{z}^0$ while $\mathbf{W}_{ij}$ retrieves the scalar element at position (i, j) within matrix $\mathbf{W}$. With this energy function, one can write out the probability that a Harmonium assigns to a data point:

$$p(\mathbf{z}^0 = \mathbf{x}) = \frac{1}{Z} \exp(-E(\mathbf{z}^0, \mathbf{z}^1))$$

where Z is the normalizing constant (or, in statistics mechanics, the partition function) needed to obtain proper probability values (and is, in fact, intractable to compute for any reasonably-sized Harmonium – fortunately, we will not need to calculate it in order to learn a Harmonium). When one works through the derivation of the gradient of the log probability $\log p(\mathbf{x})$ with respect to the synapses such as $\mathbf{W}$, they get a (contrastive) Hebbian-like update rule as follows:

$$\Delta \mathbf{W} =< \mathbf{z}^0_i \mathbf{z}^1_j >_{data} - < \mathbf{z}^0_i \mathbf{z}^1_j >_{model}$$

where the angle brackets $<>$ tell us that we need to take the expectation of the values within the brackets under a certain distribution (such as the data distribution denoted by the subscript $data$).

Technically, to compute the update above, obtaining the first term $< \mathbf{z}^0_i \mathbf{z}^1_j >_{data}$ is easy since we take the product of a data point and its corresponding hidden state under the Harmonium but obtaining $< \mathbf{z}^0_i \mathbf{z}^1_j >_{model}$ is very costly, as we would need to initialize the value of $\mathbf{z}^0$ to a random initial sate and then run a Gibbs sampler for many iterations to accurately approximate the second term. Fortunately, it was shown in work such as [3], that learning a Harmonium is still possible by replacing the term $< \mathbf{z}^0_i \mathbf{z}^1_j >_{model}$ with $< \mathbf{z}^0_i \mathbf{z}^1_j >_{recon}$, which is simply computed by using the first term's latent state $\mathbf{z}^1$ to reconstruct the input and then using this reconstruction one more to obtain its corresponding binary latent state. This is known as "Contrastive Divergence", and, although this approximation has been shown to not actual follow the gradient of any known objective function, it works well in practice when learning a generative model based on a Harmonium. Finally, the vectorized form of the Contrastive Divergence update is:

$$\Delta \mathbf{W} = \Big[(\mathbf{z}^0_{pos})^T \cdot \mathbf{z}^1_{pos} \Big] - \Big[(\mathbf{z}^0_{neg})^T \cdot \mathbf{z}^1_{neg} \Big]$$

where the first term (in brackets) is labeled as the "positive phase" (or the positive, data-dependent statistics – where $\mathbf{z}^0_{pos}$ denotes the positive phase sample of $\mathbf{z}^0$) while the second term is labeled as the "negative phase" (or the negative, data-independent statistics – where $\mathbf{z}^0_{neg}$ denotes the negative phase sample of $\mathbf{z}^0$). Note that simpler rules of a similar form can be worked out for the latent/visible bias vectors as well.

In ngc-learn, to simulate the above Harmonium generative model and its Contrastive Divergence update, we will model the positive and negative phases as simulated `NGCGraph`s, each responsible for producing the relevant statistics we need to adjust synapses. In addition, we will find that we can further re-purpose the created graphs to construct a block Gibbs sampler needed to create "fantasized" data patterns from a trained Harmonium.

9.2 Restricted Boltzmann Machines: Positive & Negative Graphs

We begin by first specifying the structure of the Harmonium system we would like to simulate. In NGC shorthand, the above positive and negative phase graphs would simply be (under one complete generative model):

```
z0 -(z0-z1)-> z1
z1 -(z1-z0) -> z0
Note: z1-z0 = (z0-z1)^T (transpose-tied synapses)
```

To construct the desired Harmonium model, particularly the structure needed to simulate Contrastive Divergence, we will need to break up the model into its key "phases", i.e., a positive phase and a negative phase. We will model each phase as its own simulated NGC graph, allowing us to craft a general approach that permits a K-step Contrastive Divergence learning process. In particular, we will use the negative graph to emulate the crucial MCMC sampling step.

Building the positive phase of our Harmonium is simple and straightforward and could be written as follows:

```
integrate_cfg = {"integrate_type" : "euler", "use_dfx" : False}
init_kernels = {"A_init" : ("gaussian",wght_sd), "b_init" : ("zeros")}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : seed}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}

## set up positive phase nodes
z1 = SNode(name="z1", dim=z_dim, beta=1, act_fx=act_fx, zeta=0.0,
             integrate_kernel=integrate_cfg, samp_fx="bernoulli")
z0 = SNode(name="z0", dim=x_dim, beta=1, act_fx="identity", zeta=0.0,
             integrate_kernel=integrate_cfg)
z0_z1 = z0.wire_to(z1, src_comp="phi(z)", dest_comp="dz_bu", cable_kernel=dcable_cfg)
z1_z0 = z1.wire_to(z0, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z0_z1,
↪"A^T"),
                   cable_kernel=dcable_cfg)
z0_z1.set_decay(decay_kernel=("l1",0.00005))

## set up positive phase update
z0_z1.set_update_rule(preact=(z0,"phi(z)"), postact=(z1,"phi(z)"), param=["A","b"])
z1_z0.set_update_rule(postact=(z0,"phi(z)"), param=["b"])

# build positive graph
print(" > Constructing Positive Phase Graph")
pos_phase = NGCGraph(K=1, name="rbm_pos")
pos_phase.set_cycle(nodes=[z0, z1]) # z0 -> z1
pos_phase.apply_constraints()
pos_phase.set_learning_order([z1_z0, z0_z1])
pos_phase.compile(batch_size=batch_size)
```

which amounts to simply simulating the projection of `z0` to latent state `z1`. The key to ensuring we simulate this simple function properly is to effectively "turn off" key parts of the neural state dynamics. Specifically, we see in the above code-snippet we set the state update `beta = 1` – this means that the full value of the deposits in `dz_bu` and `dz_bu` will be added to the current value of the compartment `z` within `z1` – and `zeta = 0` – which means that the amount of recurrent carryover is completely zeroed out (yielding a stateless node). Notice we have created a "dummy" or ghost connection via the cable `z1_z0` even though our positive phase graph will NOT actually execute the transform. However, this ghost connection is needed so that way our positive phase graph contains a visible unit bias vector (which will receive a full Hebbian update equal to the clamped visible value in the compartment `phi(z)` of `z0`).

When we trigger the `.settle()` routine for the above model given some observed data (e.g., an image or image patch), we will obtain our single-step positive phase (sufficient) statistics which include the clamped observed value of `z0 =`

x as well as its corresponding latent activity z1. This gives us half of what we need to learn a Harmonium.

To gather the rest of the statistics that we require, we need to build the negative phase of our model (to emulate its ability to "dream" up or confabulate samples from its internal model of the world). While constructing the negative phase is not that much more difficult than crafting the positive phase, it does take a bit of care to emulate the underlying "cycle" that occurs in a Harmonium when it synthesizes data when using ngc-learn's stateful dynamics. In short, we need three nodes to explicitly simulate the negative phase – a z1n_i intermediate variable that we can clamp on the positive phase value of the latent state z1, a generation output node z0n (where n labels this node as a "negative phase statistic"), and finally a generated latent state z1n that corresponds to the output node. The simulated cycle z1n_i => z0n => z1n can then be written as:

```
# set up negative phase nodes
z1n_i = SNode(name="z1n_i", dim=z_dim, beta=1, act_fx=act_fx, zeta=0.0,
              integrate_kernel=integrate_cfg, samp_fx="bernoulli")
z0n = SNode(name="z0n", dim=x_dim, beta=1, act_fx=out_fx, zeta=0.0,
              integrate_kernel=integrate_cfg, samp_fx="bernoulli")
z1n = SNode(name="z1n", dim=z_dim, beta=1, act_fx=act_fx, zeta=0.0,
              integrate_kernel=integrate_cfg, samp_fx="bernoulli")
n1_n0 = z1n_i.wire_to(z0n, src_comp="S(z)", dest_comp="dz_td", mirror_path_kernel=(z0_z1,
↪"A^T"),
                    cable_kernel=dcable_cfg) # reuse A but create new b
n0_n1 = z0n.wire_to(z1n, src_comp="phi(z)", dest_comp="dz_bu", mirror_path_kernel=(z0_z1,
↪"A+b")) # reuse A  & b
n1_n1 = z1n.wire_to(z1n_i, src_comp="z", dest_comp="dz_bu", cable_kernel=pos_scable_cfg)

# set up negative phaszupdate
n0_n1.set_update_rule(preact=(z0n,"phi(z)"), postact=(z1n,"phi(z)"), param=["A","b"])
n1_n0.set_update_rule(postact=(z0n,"phi(z)"), param=["b"])

# build negative graph
print(" > Constructing Negative Phase Graph")
neg_phase = NGCGraph(K=1, name="rbm_neg")
neg_phase.set_cycle(nodes=[z1n_i, z0n, z1n]) # z1 -> z0 -> z1
neg_phase.set_learning_order([n1_n0, n0_n1]) # forces order: c, W, b
neg_phase.compile(batch_size=batch_size)
```

where we observe that the above "negative phase" graph allows us to emulate the general K-step Contrastive Divergence algorithm (CD-K, where the commonly-used single step approximation, or K=1 is denoted as CD-1 or just "CD"). Technically, a Harmonium should be run for a very high value of K (approaching infinity) in order to obtain a proper sample from the Harmonium's equilibrium/steady state distribution. However, this would be extremely costly to simulate and, as early studies [3] observed, often only a few or even a single step of this Markov chain proved to work quite well, approximating the contrastive divergence objective (the learning algorithm's namesake) instead of direct maximum likelihood.

Notice we utilize a special helper function set_learning_order() in both the positive and negative phase graphs. This function allows us to impose an explicit order (by taking in a list of the explicit cables we have created for a particular graph) in the synaptic adjustment matrices that the NGCGraph simulation object will return (we do this to ensure that the delta matrices exactly mirror the order of those that will be returned by the positive phase graph). This is important to do when you need to coordinate the returned learning products of two or more NGCGraph objects, as we will do shortly. The order we have imposed above ensures that we return a positive delta list and a negative delta list that both respect the following ordering: db, dW, dc.

Now that we have the two graphs above, we can write the routine that will explicitly calculate the approximate synaptic weight gradients as follows:

```
x = # ... sampled data pattern (or batch of patterns) ...
Ns = x.shape[0]
## run positive phase
readouts, pos_delta = pos_phase.settle(
                        clamped_vars=[("z0","z", x)],
                        readout_vars=[("z1","S(z)")],
                        calc_delta=calc_update
                      )
z1_pos = readouts[0][2] # get positive phase binary latent state z1
## run negative phase
readouts, neg_delta = neg_phase.settle(
                        init_vars=[("z1n_i","S(z)", z1_pos)],
                        readout_vars=[("z0n","phi(z)"),("z1n","phi(z)")],
                        calc_delta=calc_update
                      )
x_hat = readouts[0][2] # return reconstruction (from negative phase)

## calculate the full Contrastive Divergence updates
delta = []
for i in range(len(pos_delta)):
    pos_dx = pos_delta[i]
    neg_dx = neg_delta[i]
    dx = ( pos_dx - neg_dx ) * (1.0/(Ns * 1.0))
    delta.append(dx) # multiply CD update by -1 to allow for minimization

opt.apply_gradients(zip(delta, pos_phase.theta))
neg_phase.set_theta(pos_phase.theta)
```

where we see that our synaptic update code carefully coordinates the positive and negative "halves" of our Harmonium by not only combining their returned local updates to compute full/final weight adjustments but also ensures that we set/point the synaptic parameters inside of the `.theta` of the negative graph to those in the `.theta` of the positive graph.

Note that one could adapt the code above (or what is found in the Model Museum `Harmonium` model structure) to emulate more advanced/powerful forms of Contrastive Divergence such as "persistent" Contrastive Divergence, where we, instead of clamping the value of `z1` to `z1n_i`, we inject random noise (or to a sample of the Harmonium's latent prior), and even an algorithm known as parallel tempering, where we would emulate multiple "negative graphs" and use samples from all of them.

Before we go and fit our Harmonium to actual data, we need to write one final bit of functionality for our model – the block Gibbs sampler to synthesize data samples given the model's current set of synaptic parameters. This is simply done as follows:

```
def sample(pos_phase, neg_phase, K, x_sample=None, batch_size=1):
    samples = []
    z1_sample = None
    ## set up initial condition for the block Gibbs sampler (use positive phase)
    readouts, _ = pos_phase.settle(
                    clamped_vars=[("z0","z", x_sample)],
                    readout_vars=[("z1","S(z)")],
                    calc_delta=False
                  )
    z1_sample = readouts[0][2]
    pos_phase.clear()
```

(continues on next page)

(continued from previous page)

```
    ## run block Gibbs sampler to generate a chain of sampled patterns
    neg_phase.inject([("z1n_i", "S(z)", z1_sample)]) # start chain at sample
    for k in range(K):
        readouts, _ = neg_phase.settle(
                        readout_vars=[("z0n", "phi(z)"), ("z1n", "phi(z)")],
                        calc_delta=False, K=1
                      )
        z0_prob = readouts[0][2] # the "sample" of z0
        z1_prob = readouts[1][2]
        samples.append(z0_prob) # collect output sample
        neg_phase.clear()
        neg_phase.inject([("z1n_i", "phi(z)", z1_prob)])
    return samples
```

Notice that this sampling function produces a list/array of samples in the order in which they were produced by the Markov chain constructed above.

9.3 Using the Harmonium to Dream Up Handwritten Digits

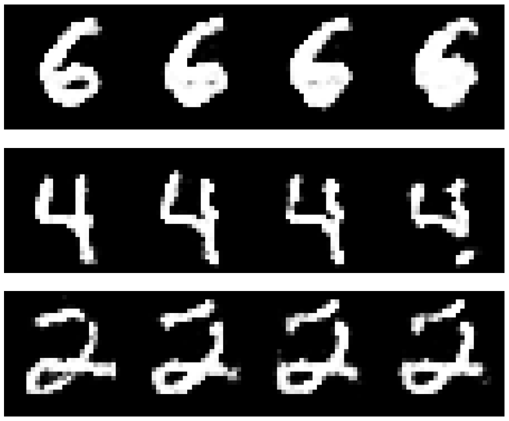
We finally take the Harmonium that we have constructed above and fit it to some MNIST digits (the same dataset we used in Walkthrough #1). Specifically, we will leverage the *Harmonium*, model in the Model Museum as it implements the above core components/functions internally. In the
script `sim_train.py`, you will find a general simulated training loop similar to what we have developed in previous walkthroughs that will fit our Harmonium to the MNIST database (unzip the file `mnist.zip` in the `/walkthroughs/data/` directory if you have not already) by cycling through it several times, saving the final (best) resulting to disk within the `rbm/` sub-directory. Go ahead and execute the training process as follows:

```
$ python sim_train.py --config=rbm/fit.cfg --gpu_id=0
```

which will fit/adapt your Harmonium to MNIST. Once the training process has finished, you can then run the following to sample from Harmonium using block Gibbs sampling:

```
$ python sample_rbm.py --config=rbm/fit.cfg --gpu_id=0
```

which will take your trained Harmonium's negative phase and use it to synthesize some digits. You should see inside the `rbm/` sub-directory something similar to:

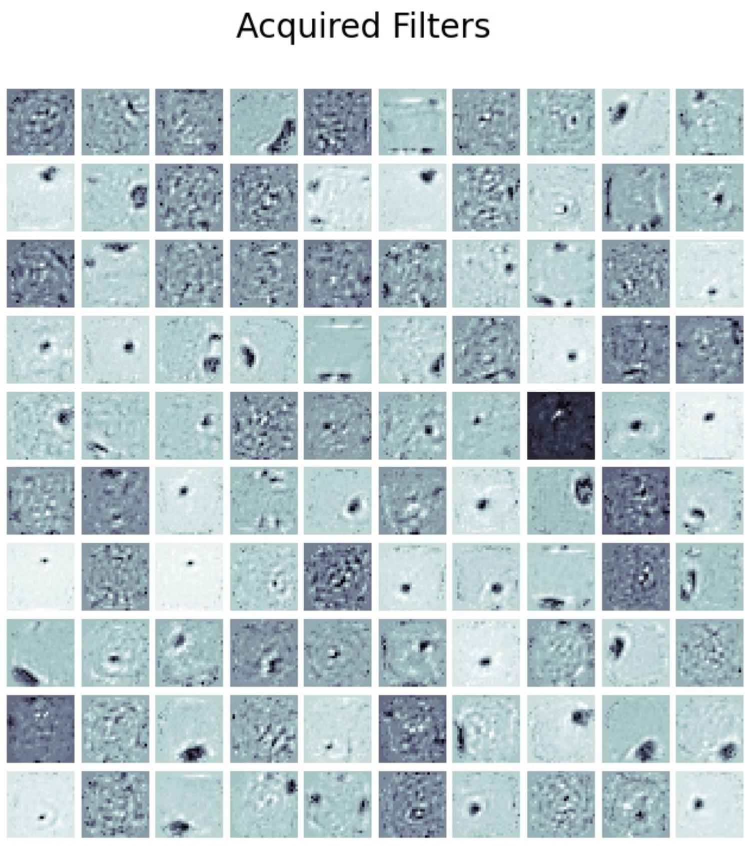


It is important to understand that the three rows of samples shown above come from particular points in the block Gibbs sampling process. Specifically, the script that you ran sets the number of steps to be `K=80` and stores/visualizes a fantasized image every `8` steps. Furthermore, we initialize each of the three above Markov chains with a randomly sampled image from the MNIST training dataset. Note that higher-quality samples can be obtained if one modifies the earlier Harmonium to learn with persistent Contrastive Divergence or parallel tempering.

Finally, you can also run the `viz_filters.py` script to extract the acquired filters/ receptive fields of your trained Harmonium, much as we did for the sparse coding and hierarchical ISTA models in Walkthroughs #4 and #5, as follows:

```
$ python viz_filters.py --config=rbm/fit.cfg --gpu_id=0
```

to obtain a plot similar to the one below:



Interestingly enough, we see that our Harmonium has extracted what appears to be rough stroke features, which is what it uses when sampling its binary latent feature detectors to compose final synthesized image patterns (each binary feature detector serves as Boolean function that emits a `1` if the feature/filter is to be used and a `0` if not). In particular, notice that the filters our Harmonium has acquired are a bit more prominent due to the weight decay we applied earlier via `z0_z1.set_decay(decay_kernel=("l1",0.00005))` (which tells the `NGCGraph` simulation object to apply Laplacian/L1 decay to the `W` matrix of our RBM).

On a final note, the Harmonium we have built in this walkthrough is a classical Bernoulli Harmonium and thus assumes that the input data features are binary in nature. If one wants to model data that is continuous/real-valued, then the Harmonium model above would need to be adapted to utilize visible units that follow a distribution such as the multivariate Gaussian distribution, yielding, for example, a Gaussian restricted Boltzmann machine (GRBM).

9.4 References

[1] Smolensky, P. "Information Processing in Dynamical Systems: Foundations of Harmony Theory." Parallel distributed processing: explorations in the microstructure of cognition 1 (1986). [2] Geoffrey Hinton. Products of Experts. International conference on artificial neural networks (1999). [3] Hinton, Geoffrey E. "Training products of experts by maximizing contrastive likelihood." Technical Report, Gatsby computational neuroscience unit (1999).

CHAPTER

TEN

WALKTHROUGH 7: SPIKING NEURAL NETWORKS

In this demonstration, we will design a three layer spiking neural network (SNN). We will specifically cover the special base spiking node classes within ngc-learn's nodes-and-cables system, particularly examining the properties of the leaky integrate-and-fire (LIF) node with respect to modeling voltage and spike trains. In addition, we will cover how to set up a simple online learning process for training the SNN on the MNIST database. After going through this demonstration, you will:

1. Learn how to use/setup the `SpNode_LIF` (the LIF node class) and the `SpNode_Enc` (the Poisson spike train node class) and visualize the voltage as a function of input current and the resulting spikes in a raster plot.
2. Build a spiking network using the `SpNode_LIF` and the `SpNode_Enc` nodes and simulate its adaptation to MNIST image patterns by setting up a simple algorithm known as broadcast feedback alignment.

Note that the folders of interest to this demonstration are:

- `walkthroughs/demo7/`: this contains the necessary simulation scripts
- `walkthroughs/data/`: this contains the zipped copy of the digit image arrays

10.1 Encoding Data Patterns as Poisson Spike Trains

Before we start crafting a spiking neural network (SNN), let us first turn our attention to the data itself. Currently, the patterns in the MNIST database are in continuous/real-valued form, i.e., pixel values normalized to the range of $[0, 1]$. While we could directly use them as input into a network of LIF neurons, as was done in [1] (meaning we would copy the literal data vector each step in time, much as we have done in previous walkthroughs), it would be better if we could first convert them to binary spike trains themselves given that SNNs are technically meant to process time-varying information. While there are many ways to encode the data as spike trains, we will take the simplest approach in this walkthrough and work with an encoding scheme known as rate encoding.

Specifically, rate encoding entails normalizing the original real-valued data vector $\mathbf{x}$ to the range of $[0, 1]$ and then treating each dimension $\mathbf{x}_i$ as the probability that a spike will occur, thus yielding (for each dimension) a rate code with a value of $\mathbf{s}_i$. In other words, each feature drives a Bernoulli distribution of the form where $\mathbf{s}_i \sim \mathcal{B}(n, p)$ where $n = 1$ and $p = \mathbf{x}_i$. This, over time, results in a Poisson process where the rate of firing is dictated by solely in proportion to a feature's value.

To rate code your data, let's start by using a simple function in ngc-learn's `ngclearn.utils.stat_utils` module. Assuming we have a simple 10-dimensional data vector $\mathbf{x}$ (of shape `1 x 10`) with values in the range of $[0, 1]$, we can convert it to a spike train over 100 steps in time as follows:

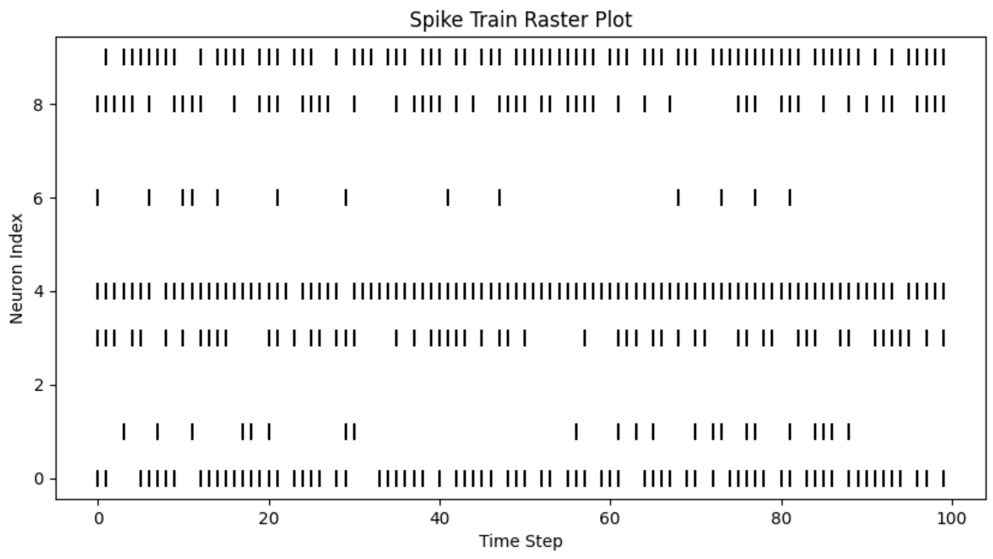
```
import tensorflow as tf
import numpy as np
from ngclearn.utils import stat_utils as stat
import ngclearn.utils.viz_utils as viz
```

(continues on next page)

(continued from previous page)

```
seed = 1990
tf.random.set_seed(seed=seed)
np.random.seed(seed)

z = np.zeros((1,10),dtype=np.float32)
z[0,0] = 0.8
z[0,1] = 0.2
z[0,3] = 0.55
z[0,4] = 0.9
z[0,6] = 0.15
z[0,8] = 0.6
z[0,9] = 0.77
spikes = None
for t in range(100):
    s_t = stat.convert_to_spikes(z, gain=1.0)
    if t > 0:
        spikes = tf.concat([spikes, s_t],axis=0)
    else:
        spikes = s_t
spikes = spikes.numpy()

viz.create_raster_plot(spikes, s=100, c="black")
```

where we notice that in the first dimension `[0,0]`, fifth dimension `[0,4]`, and the final dimension `[0,9]` set to fairly high spike probabilities. This code will produce and save locally to disk the following raster plot for visualizing the resulting spike trains:



where we see that the first, middle/fifth, and tenth dimensions do indeed result in denser spike trains. Raster plots are simple visualization tool in computational neuroscience for examining the trial-by-trial variability of neural responses and allow us to graphically examine timing (or the frequency of firing), one of the most important aspects of neuronal action potentials/spiking patterns. Crucially notice that the function ngc-learn offers for converting to Poisson spike trains does so on-the-fly, meaning that you can generate binary spike pattern vectors from a particular normalized real-valued vector whenever you need to (this facilitates online learning setups quite well). If you examine the API for the `convert_to_spikes()` routine, you will also notice that you can also control the firing frequency further with the `gain` argument – this is useful for recreating certain input spike settings report in certain computational neuroscience publications. For example, with MNIST, it is often desired that the input firing rates are within the approximate range of 0 to 63.75 Hertz (Hz) (as in [2]) and this can easily be recreated for data normalized to $[0, 1]$ by setting the `gain` parameter to `0.25` (we will also do this for this walkthrough for the model we will build later).

Note that another method offered by ngc-learn for converting your real-valued data vectors to Poisson spike trains is through the *SpNode_Enc*. This node is a convenience node that effectively allows us to do the same thing as the code snippet above (for example, upon inspecting its API, you will see an argument to its constructor is the `gain` that you can set yourself). However, the `SpNode_Enc` conveniently allows the spike encoding process to be directly integrated into the `NGCGraph` simulation
object that you will ultimately want to create (as we will do later in this walkthrough). Furthermore, internally, this node provides you with some useful optional compartments that are calculated during simulation such as variable traces/filters.

10.2 The Leaky Integrate-and-Fire Node

Now that we have considered how to transform our data into Poisson spike trains for use with an SNN we can move on to building the SNN itself. One of the core nodes offered by ngc-learn to do this is the *SpNode_LIF*, or the leaky integrate-and-fire (LIF) node (also referred to as the leaky integrator in some papers). This node has quite a few compartments and constants but only a handful are important for understanding how this model governs spiking/firing rates during an `NGCGraph`'s simulation window. Specifically, in this walkthrough, we will examine the following compartments – `dz_bu, dz_td, Jz, Vz, Sz, ref` formally labeled as $\mathbf{dz}_{bu}$, $\mathbf{dz}_{td}$, $\mathbf{j}_t$, $\mathbf{v}_t$, $\mathbf{s}_t$, and $\mathbf{r}_t$ (the subscript t indicates that this compartment variable takes on a certain value at a certain time step t)– and the following constants – `V_thr, dt, R, C, tau_m, ref_T` formally labeled as V_{thr}, Δt, R, C, τ_m, and T_{ref}. (The other compartments and constants that we do not cover here are useful for more advanced simulations/other situations and will be discussed in future tutorials/walkthroughs.)

Now let us unpack this node by first defining the compartments:

- $\mathbf{dz}_{bu}$ and $\mathbf{dz}_{td}$ are where signals from external sources (such as other nodes) are to be deposited (much like the state node *SNode*) - note these signals contribute directly to the electrical current of the neurons within this node
- $\mathbf{j}_t$: the current electrical current of the neurons within this node (specifically computed in this node's default state as $\mathbf{j}_t = \mathbf{dz}_{bu} + \mathbf{dz}_{td}$)
- $\mathbf{v}_t$: the current membrane potential of the neurons within this node
- $\mathbf{S}_t$: the current recording/reading of any spikes produced by this node's neurons
- $\mathbf{r}_t$: the current value of the absolute refractory variables - this accumulates with time (and forces neurons to rest)

and finally the constants:

- V_{thr}: threshold that a neuron's membrane potential must overcome before a spike is transmitted
- Δt: the integration time constant, on the order of milliseconds (ms)
- R: the neural (cell) membrane resistance, on the order of mega Ohms ($M\Omega$)
- C: the neural (cell) membrane capacitance, on the order of picofarads (pF)
- τ_m: membrane potential time constant (also $\tau_m = R * C$ - resistance times capacitance)
- T_{ref}: the length of a neuron's absolute refractory period

With above defined, we can now explicitly lay out the underlying (linear) ordinary differential equation that the `SpNode_LIF` evolves according to:

$$\tau_m \frac{\partial \mathbf{v}_t}{\partial t} = (-\mathbf{v}_t + R\mathbf{j}_t), \text{ where, } \tau_m = RC$$

and with some simple mathematical manipulations (leveraging the method of finite differences), we can derive the Euler integrator employed by the SpNode_LIF as follows:

$$\tau_m \frac{\partial \mathbf{v}_t}{\partial t} = (-\mathbf{v}_t + R\mathbf{j}_t)$$
$$\tau_m \frac{\mathbf{v}_{t+\Delta t} - \mathbf{v}_t}{\Delta t} = (-\mathbf{v}_t + R\mathbf{j}_t)$$
$$\mathbf{v}_{t+\Delta t} = \mathbf{v}_t + (-\mathbf{v}_t + R\mathbf{j}_t)\frac{\Delta t}{\tau_m}$$

where we see that above integration tells us that the membrane potential of this node varies over time as a function of the sum of its input electrical current $\mathbf{j}_t$ (multiplied by the cell membrane resistance) and a leak (or decay) $-\mathbf{v}_t$ modulated by the integration time constant divided by the membrane time constant. The SpNode_LIF allows you to control the value of τ_m either directly (and will tell the node to set $R = 1$ and $C = \tau_m$ and the node will ignore any argument values provided for R and C) or via R and C. (Notice that this default state of the SpNode_LIF assumes that the input spike signals from external nodes that feed into $\mathbf{dz}_{bu}$ and $\mathbf{dz}_{td}$) result in an instantaneous jump in each neuron's synaptic current $\mathbf{J}_t$ but this assumption/simplification can be removed by setting SpNode_LIF's argument zeta to any non-zero value in order to tell the node that it needs to integrate its synaptic current over time - we will not, however, cover this functionality in this walkthrough.)

In effect, given the above, every time the SpNode_LIF's .step() function is called within an NGCGraph simulation object, the above Euler integration of the membrane potential differential equation is happening each time step. Knowing this, the last item required to understand ngc-learn's LIF node's computation is related to its $\mathbf{s}_t$. The spike reading is computed simply by comparing the current membrane potential $\mathbf{v}_t$ to the constant threshold defined by V_{thr} according to the following piecewise function:

$$\mathbf{s}_{t,i} = \begin{cases} 1 & \mathbf{v}_{t,i} > V_{thr} \\ 0 & \text{otherwise.} \end{cases}$$

where we see that if the ith neuron's membrane potential exceeds the threshold V_{thr}, then a voltage spike is emitted. After a spike is emitted, the ith neuron within the node needs to be reset to its resting potential and this done with the final compartment we mentioned, i.e., the refractory variable $\mathbf{r}_t$. The refractory variable $\mathbf{r}_t$ is important for hyperpolarizing the ith neuron back to its resting potential (establishing a critical reset mechanism – otherwise, the neuron would fire out of control after overcoming its threshold) and reducing the amount of spikes generated over time. This reduction is one of the key factors behind the power efficiency of actual neuronal systems. Another aspect of ngc-learn's refractory variable is the temporal length of the reset itself, which is controlled by the T_{ref} (T_ref) constant – this yields what is known as the absolute refractory period, or the interval of time at which a second action potential absolutely cannot be initiated. If T_{ref} is set to be greater than zero, then the ith neuron that fires will be forced to remain at its resting potential of zero for the duration of this refractory period.

Now that we understand the key compartments and constants inherent to an LIF node, we can start simulating one. Let us visualize the spiking pattern of our LIF node by feeding into it a step current, where the electrical current starts at 0 then switches to 0.3 at $t = 10$ (ms). Specifically, we can plot the input current, the neuron's voltage, and its output spikes as follows:

```
import os
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# import general simulation utilities
import ngclearn.utils.viz_utils as viz
from ngclearn.engine.nodes.spnode_lif import SpNode_LIF
from ngclearn.engine.ngc_graph import NGCGraph
```

(continues on next page)

(continued from previous page)

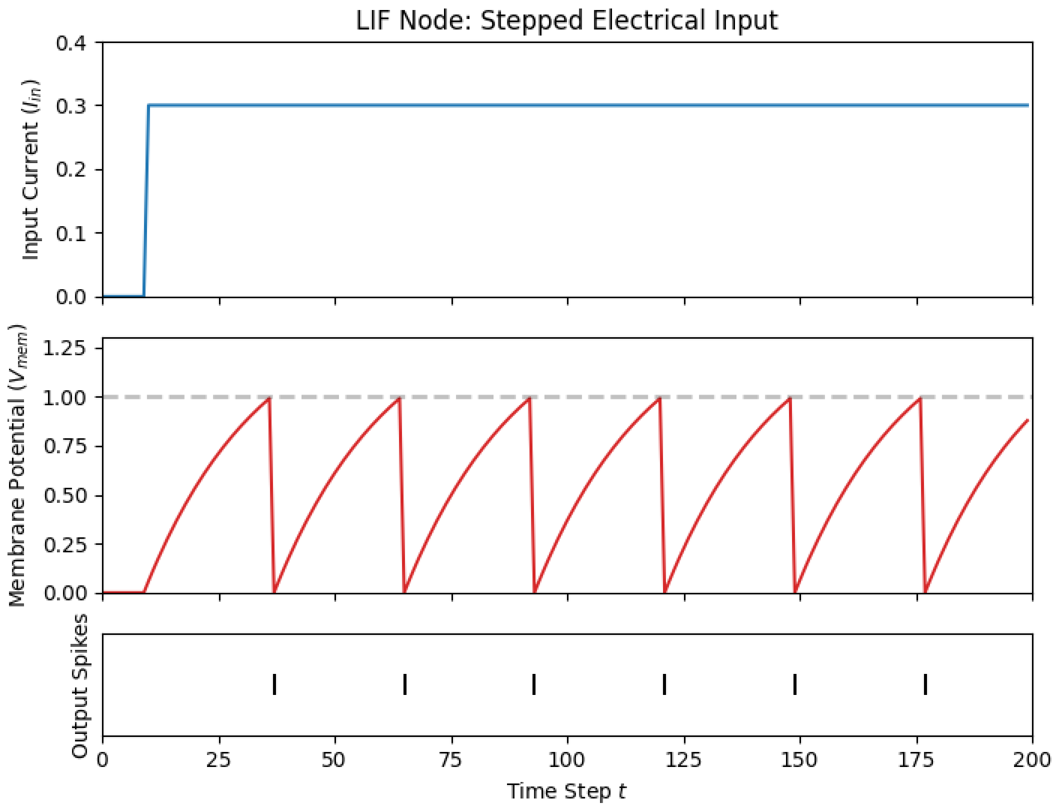
```
seed = 1990 # 69
os.environ["CUDA_VISIBLE_DEVICES"]="0"
tf.random.set_seed(seed=seed)
np.random.seed(seed)

dt = 1e-3 # integration time constant
R = 5.0 # in mega Ohms
C = 5e-3 # in picofarads
V_thr = 1.0
ref_T = 0.0 # length of absolute refractory period
leak = 0
beta = 0.1
z1_dim = 1
integrate_cfg = {"integrate_type" : "euler", "dt" : dt}
# set refractory period to be really short
spike_kernel = {"V_thr" : V_thr, "ref_T" : ref_T, "R" : R, "C" : C}
trace_kernel = {"dt" : dt, "tau" : 5.0}

# set up a single LIF node system
lif = SpNode_LIF(name="lif_unit", dim=z1_dim, integrate_kernel=integrate_cfg,
                 spike_kernel=spike_kernel, trace_kernel=trace_kernel)
model = NGCGraph()
model.set_cycle(nodes=[lif])
info = model.compile(batch_size=1, use_graph_optim=False)

# create a synthetic electrical step current
current = tf.concat([tf.zeros([1,10]),tf.ones([1,190])*0.3],axis=1)

curr_rec = []
voltage = []
refract = []
spike_train = []

# simulate the LIF node
model.set_to_resting_state()
for t in range(current.shape[1]):
    I_t = tf.ones([1,1]) * current[0,t]
    curr_rec.append(float(I_t))
    model.clamp([("lif_unit", "Jz", I_t)])
    model.step(calc_delta=False)
    J_t = model.extract("lif_unit", "Jz")
    V_t = model.extract("lif_unit", "Vz")
    S_t = model.extract("lif_unit", "Sz")
    ref = model.extract("lif_unit", "ref")

    refract.append(float(ref))
    voltage.append(float(V_t))
    spike_train.append(float(S_t))

cur_in = np.asarray(curr_rec)
mem_rec = np.asarray(voltage)
spk_rec = np.asarray(spike_train)
```

(continues on next page)

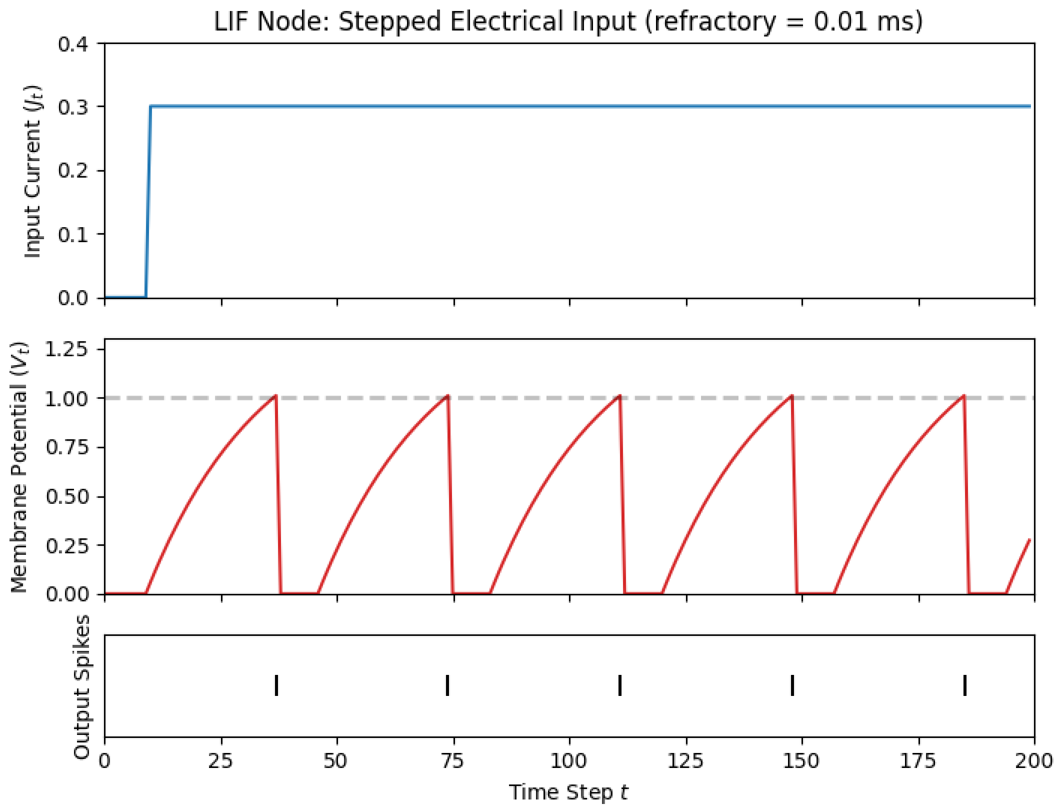
(continued from previous page)

```
viz.plot_lif_neuron(cur_in, mem_rec, spk_rec, refract, dt, thr_line=V_thr, max_mem_val=1.
↪3,
                    title="LIF Node: Stepped Electrical Input")
```

which produces the following plot (saved as `lif_plot.png` locally to disk):



where we see that, given a build-up over time in the neuron's membrane potential (since the current is constant and non-zero after 10 ms), a spike is emitted once the value of the membrane potential exceeds the threshold (indicated by the dashed horizontal line in the middle plot) $V_{thr} = 1$. Notice that if we play with the value of `ref_T` (the refactory period T_{ref}) and change it to something like `ref_T = 10 * dt` (ten times the integration time constant), we get the following plot:



where we see that after the LIF neuron fires, it remains stuck at its resting potential for a period of 0.01 ms (the short flat periods in the red curve starting after the first spike).

10.3 Learning a Spiking Network with Broadcast Alignment

Now that we examined the `SpNode_Enc` and analyzed a single `SpNode_LIF`, we can now finally build a complete SNN model to simulate. Building the SNN is no different than any other system in ngc-learn and is done as follows (note that the settings shown below follow closely those reported in [2]):

```
x_dim = # dimensionality of input data
z_dim = # number of neurons for the internal layer
y_dim = # dimensionality of output space (or number of classes)
dt = 0.25 # integration time constant
tau_mem = 20 # membrane potential time constant
V_thr = 0.4 # spiking threshold
# Default for rec_T of 1 ms will be used - this is the default for SpNode_LIF(s)
integrate_cfg = {"integrate_type" : "euler", "dt" : dt}
spike_kernel = {"V_thr" : V_thr, "tau_mem" : tau_mem}
trace_kernel = {"dt" : dt, "tau" : 5.0}

# set up system -- notice for z2, a gain of 0.25 yields spike frequency of 63.75 Hz
z2 = SpNode_Enc(name="z2", dim=x_dim, gain=0.25, trace_kernel=trace_kernel)
mu1 = SNode(name="mu1", dim=z_dim, act_fx="identity", zeta=0.0)
```

(continues on next page)

(continued from previous page)

```
z1 = SpNode_LIF(name="z1", dim=z_dim, integrate_kernel=integrate_cfg,
                spike_kernel=spike_kernel, trace_kernel=trace_kernel)
mu0 = SNode(name="mu0", dim=y_dim, act_fx="identity", zeta=0.0)
z0 = SpNode_LIF(name="z0", dim=y_dim, integrate_kernel=integrate_cfg,
                spike_kernel=spike_kernel, trace_kernel=trace_kernel)
e0 = ENode(name="e0", dim=y_dim)
t0 = SNode(name="t0", dim=y_dim, beta=beta, integrate_kernel=integrate_cfg, leak=0.0)
d1 = FNode_BA(name="d1", dim=z_dim, act_fx="identity") # BA teaching node

# create cable wiring scheme relating nodes to one another
init_kernels = {"A_init" : ("gaussian", wght_sd), "b_init" : ("zeros",)}
dcable_cfg = {"type": "dense", "init_kernels" : init_kernels, "seed" : seed}
pos_scable_cfg = {"type": "simple", "coeff": 1.0}
neg_scable_cfg = {"type": "simple", "coeff": -1.0}

z2_mu1 = z2.wire_to(mu1, src_comp="Sz", dest_comp="dz_td", cable_kernel=dcable_cfg)
mu1.wire_to(z1, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=pos_scable_cfg)
z1_mu0 = z1.wire_to(mu0, src_comp="Sz", dest_comp="dz_td", cable_kernel=dcable_cfg)
mu0.wire_to(z0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=pos_scable_cfg)
z0.wire_to(e0, src_comp="Sz", dest_comp="pred_mu", cable_kernel=pos_scable_cfg)
t0.wire_to(e0, src_comp="phi(z)", dest_comp="pred_targ", cable_kernel=pos_scable_cfg)
e0.wire_to(t0, src_comp="phi(z)", dest_comp="dz_td", cable_kernel=neg_scable_cfg)
```

this sets up an SNN structure with three layers – an input layer z2 containing the Poisson spike train nodes (which will be driven by input data x), an internal layer of LIF nodes, and an output layer of LIF nodes. We have also opted to simplify the choice of meta-parameters and directly set the membrane potential constant tau_mem directly (instead of messing with membrane resistance and capacitance). Nothing else is out of the ordinary in creating an NGCGraph except that we have also included a simple specialized convenience node d1, which will serve as a special part of our SNN structure that will naturally give us an easy way to adapt this SNN's parameters with an online learning process. This convenience node is not all too different from a forward node (FNode) except it has been adapted to a sort of "teaching node" format that effectively takes in its input signals in its dz compartment and multiplicatively combines them with the special approximate derivative (or dampening) function developed in [1] (see *FNode_BA* for the details of this special dampening function).

With the above nodes and cables set up all the remains is to define the SNN's synaptic learning/adjustment rules. Given our special teaching node d1, we can directly construct a simple learning scheme based on an algorithm known as broadcast alignment (BA) [1], which, in short, posits that a special set of error feedback synapses (that a randomly initialized and never adjusted during simulation) can directly transform and carry error signals from a particular spot (such as the output layer of an SNN) back to any internal layer as needed. These backwards transmitted signals only need to travel down a very short feedback pathway and the outputs of these randomly projected error signals (when combined with the special dampening function mentioned above) can serve as powerful teaching signals to drive change in synaptic efficacy. To craft the BA approach to learning an SNN, we can then utilize the feedback pathway created by d1 to drive learning through ngc-learn's typical Hebbian updates as shown below:

```
# set up the SNN update rules and make relevant edges aware of these
from ngclearn.engine.cables.rules.hebb_rule import HebbRule

rule1 = HebbRule() # create a local weighted Hebbian rule for internal layer
rule1.set_terms(terms=[(z2,"z"), (d1,"phi(z)")], weights=[1.0, (1.0/(x_dim * 1.0))])
z2_mu1.set_update_rule(update_rule=rule1, param=["A", "b"])

rule2 = HebbRule() # create a local weighted Hebbian rule for output layer
rule2.set_terms(terms=[(z1,"Sz"), (e0,"phi(z)")], weights=[1.0, (1.0/(z_dim * 1.0))])
```

(continues on next page)

(continued from previous page)

```
z1_mu0.set_update_rule(update_rule=rule2, param=["A", "b"])
```

where we notice two special things that the above code is doing in contrast to prior walkthroughs: 1) we have exposed the lower-level local rule system of ngc-learn which allows the user/experimenter to define their own custom local updates if needed (the default in ngc-learn is a simple two-term, unweighted Hebbian adjustment rule, which is what you have been using in all prior walkthroughs without knowing it), and 2) we have modified the typical Hebbian update rule to be a weighted Hebbian update by setting the weights of the post-activation terms to be a function of the number of pre-synaptic neurons for a given layer (this is akin to a layer-wise learning rate and provided a simple means of initializing the step size for gradient descent as in [1]).

With the learning rules, we can continue as normal and initialize and compile our the `NGCGraph` for our desired SNN as follows:

```
# Set up graph - execution cycle/order
model = NGCGraph(name="snn_ba")
model.set_cycle(nodes=[z2, mu1, z1, mu0, z0, t0])
model.set_cycle(nodes=[e0])
model.set_cycle(nodes=[d1])
info = model.compile(batch_size=batch_size)
opt = tf.keras.optimizers.SGD(1.0) # create an SGD optimization rule with no learning␣
↪rate
```

only noting that, because we have set our `NGCGraph` to use weighted Hebbian updates, we do not need to specify a learning rate for our stochastic gradient descent optimization rule (as the learning rates are baked into the Hebbian rules now).

The last item we would like to cover is how specifically the SNN system we have created above will be simulated. Normally, after building an `NGCGraph`, you would generally use its `.settle()` function to simulate the processing of data over a window of time (of length `K`). Although you could technically do this with the SNN too, since the SNN is meant to learn online across a spike train, it is simpler and more appropriate to use ngc-learn's lower-level online simulation API (which was discussed in *Tutorial #1*). This will specifically allow us to integrate our SGD optimization rule directly into and extract some special statistic from the step-by-step evolution process of our SNN as it processes data-driven Poisson spike trains. The code we would need to do this is below:

```
x = # input pattern vector/matrix (real-valued & normalized to [0,1])
y = # labels/one-hot encodings associated with x

T = 100 # ms (length of simulation window)

model.set_to_resting_state() # set all neurons to their resting potentials
y_count = y * 0
y_hat = 0.0
for t in range(T):
    model.clamp([("z2", "z", x), ("t0", "z", y_)])
    delta = model.step(calc_delta=True)
    y_hat = model.extract("z0", "Jz") + y_hat
    y_count += model.extract("z0", "Sz")
    if delta is not None:
        opt.apply_gradients(zip(delta, model.theta)) # update synaptic efficacies
model.clear() # clear simulation object memory
# compute approximate Multinoulli distribution
y_hat = tf.nn.softmax(y_hat/T)
# get predicted labels from spike counts
y_pred = tf.cast(tf.argmax(y_count,1),dtype=tf.int32)
```

where we see that we have explicitly designed the simulation loop by hand, giving us the flexibility to introduce the synaptic updates at each time step. Notice that we also added in some extra statistics `y_hat` and `y_count` – `y_hat` is the approximate label distribution produced by our SNN over a stimulus window of `T = 100` milliseconds and `y_count` stores the spike counts (one per class label/output node) for us to finally extract the model's global predicted labels (by taking the argmax of `y_count` to get `y_pred`).

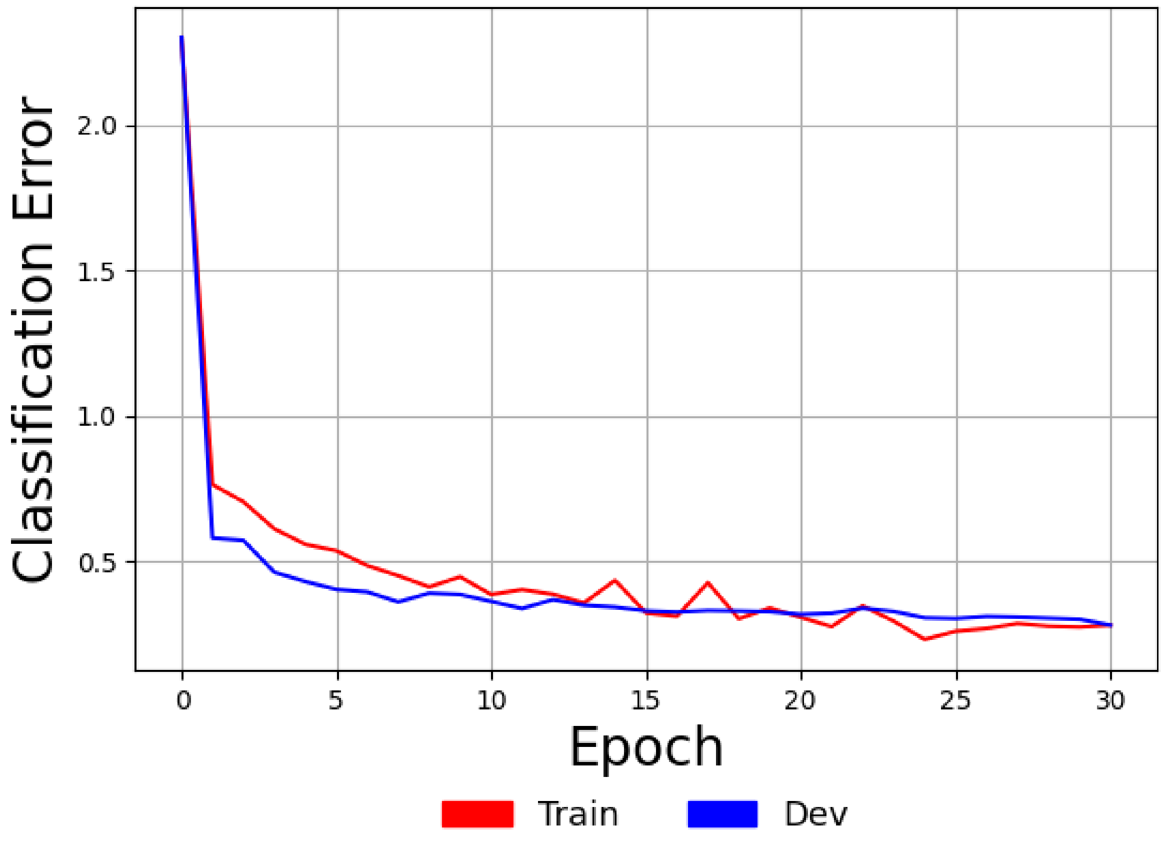
The above code (as well as a few extra convenience utilities/wrapper functions) has been integrated into the Model Museum as the official *SNN_BA*, which is the model that is imported for you in the provided `train_sim.py` script (Note: unzip the file `mnist.zip` in the `/walkthroughs/data/` directory if you have not already.) Go ahead and run `train_sim.py` as follows:

```
$ python sim_train.py --config=snn/fit.cfg --gpu_id=0
```

which will simulate the training of your SNN-BA on the MNIST database for 30 epochs. This script will save your trained SNN model to the `/snn/` sub-directory from which you can then run the evaluation script (which simply simulates your trained SNN on the MNIST test set but with the synaptic adjustment turned off). You should get an output similar to the one below:

```
Ly = 1.4924614429473877  Acc = 0.9684000015258789
```

meaning that our three-layer SNN has nearly reached 97% test classification accuracy (recall that we are counting spikes and, for each row in an evaluated test mini-batch matrix, the output LIF node with highest spike count at the end of `100` ms is chosen as the SNN's predicted label). This evaluation script also generates and saves to the `/snn/` sub-directory a learning curve plot (recorded during the simulated training for both the training and development data subsets) shown below:



where we see that the SNN has decreased its approximate negative log likelihood from a starting point of about `2.30` nats to nearly `0.28` nats. This is bearing in mind that we have estimated class probabilities output by our SNN by probing and averaging over electrical current values from `100` simulated milliseconds per test pattern mini-batch. We remark that this constructed SNN is not particularly deep and with additional layers of `SpNode_LIF` nodes, improvements to its accuracy and approximate log likelihood would be possible (the BA learning approach would, in principle, work well for any number of layers). This is motivated by the results reported in [1], where additional layers were found to improve generalization a bit more and, as reported in [2], using layers with many more LIF neurons were demonstrated to boost predictive accuracy (with nearly `6400` LIF neurons).

With that, you have now walked through the process of constructing a full SNN and fit it to an image pattern dataset. Note that our `SNN_BA` offers a bit more functionality than the SNN designed in [1] given that ours directly processes Poisson spike trains while the one in [1] focused on processing the raw real-valued pattern vectors (copying the input `x` to each time step). Furthermore, our SNN processing loop usefully approximates an output distribution by averaging over electrical current inputs (allowing us to measure its predictive log likelihood).

There is certainly more to the story of spike trains far beyond the model of leaky integrate-and-fire neurons and Poisson spike train encoding. Notably, there are many, many more neurobiological details that this type of modeling omits and one exciting continual development of ngc-learn is to continue to incorporate and test its dynamics simulator on an ever-increasing swath of spike-based nodes of increasing complexity and biological faithfulness, such as the Hodgkin–Huxley model [3], as well as other learning mechanisms it is capable of simulating, such as spike-timing-dependent plasticity (or STDP, which will be discussed in later tutorials/walkthroughs) as was used in [2]).

[1] Samadi, Arash, Timothy P. Lillicrap, and Douglas B. Tweed. "Deep learning with dynamic spiking neurons and fixed feedback weights." Neural computation 29.3 (2017): 578-602. [2] Diehl, Peter U., and Matthew Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity." Frontiers in computational neuroscience 9 (2015): 99. [3] Hodgkin, Alan L., and Andrew F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." The Journal of physiology 117.4 (1952): 500.

CHAPTER

ELEVEN

THE MODEL MUSEUM

Predictive processing has undergone many important developments over the decades, dating back to Hermann von Helmholtz's theory of "unconscious inference" in perception which itself operationalized the ideas of the 18th century philosopher Immanuel Kant. It has risen as a promising theoretical and mathematical model of various aspects of neural circuitry in computational neuroscience, serving as one embodiment of the Bayesian brain hypothesis, and has been shown to be a powerful computational modeling tool for cognitive science and statistical/machine learning. Many different architectures/systems, often designed to serve one or a few particular modeling purposes, have been (and still are being) proposed. Given this, one of ngc-learn's aims is to capture an approximate snapshot of as many of these architectures/ideas as possible.

Given the generality of the NGC computational framework [1], many flavors of predictive processing can be recovered/derived and it is within ngc-learn's Model Museum that we intend to model and preserve variant models (as they historically have been and currently are being created). This allows current and future scientists, engineers, and enthusiasts to examine these models, much as one would curiously examine exhibits. such as paintings or preserved mechanical inventions and technological artifacts, at a museum. The Model Museum also provides an opportunity for those working in the domain of predictive processing to publish their successful structures/ideas that are presented in publications and/or tested applications (contact us if you have a particular published or representative predictive processing model that you would like exhibited and to be integrated into the Model Museum for the benefit of the community). In parallel, since ngc-learn is an evolving library, we will be working to curate and update the museum with representative models over time and several are already under development/testing (stay tuned for their release across software releases/patches/edits).

As mentioned above, NGC predictive processing models have historically been designed to serve particular purposes, thus we wrap their underlying NGC graphs in an agent structure that provides particular documented convenience functions that allow the user/modeler to interact with such models according to their intended purpose/use. For example, a published/public NGC model that was developed to classify data will offer functionality for categorization in a relevant prediction routine while another one that was created to operate as a generative/density estimator will sport a routine(s) for sampling/synthesization.

Current models that we have implemented in the Model Museum so far include:

1. GNCN-t1/Rao - the model proposed in (Rao & Ballard, 1999) [2]
2. GNCN-t1-Sigma/Friston - the model proposed in (Friston, 2008) [3]
3. GNCN-PDH - the model proposed in (Ororbia & Kifer, 2022) [1]
4. GNCN-t1-FFM - the model developed in (Whittington & Bogacz, 2017) [4]
5. GNCN-t1-SC - the model proposed in (Olshausen & Field, 1996) [5]
6. Harmonium - the model developed in (Smolensky, 1986; Hinton 1999) [6] [7]
7. SNN-BA - a generalization of the spiking model in (Samadi et al., 2017) [8]

(If there is a model you think should be exhibited/integrated into the Model Museum, and/or would like to contribute, please write us at ago@cs.rit.edu or raise a github issue.)

References: [1] Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022). [2] Rao, Rajesh PN, and Dana H. Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects." Nature neuroscience 2.1 (1999): 79-87. [3] Friston, Karl. "Hierarchical models in the brain." PLoS Computational Biology 4.11 (2008): e1000211. [4] Whittington, James CR, and Rafal Bogacz. "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity." Neural computation 29.5 (2017): 1229-1262. [5] Olshausen, B., Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996). [6] Hinton, Geoffrey E. "Training products of experts by maximizing contrastive likelihood." Technical Report, Gatsby computational neuroscience unit (1999). [7] Smolensky, P. "Information Processing in Dynamical Systems: Foundations of Harmony Theory." Parallel distributed processing: explorations in the microstructure of cognition 1 (1986). [8] Samadi, Arash, Timothy P. Lillicrap, and Douglas B. Tweed. "Deep learning with dynamic spiking neurons and fixed feedback weights." Neural computation 29.3 (2017): 578-602.

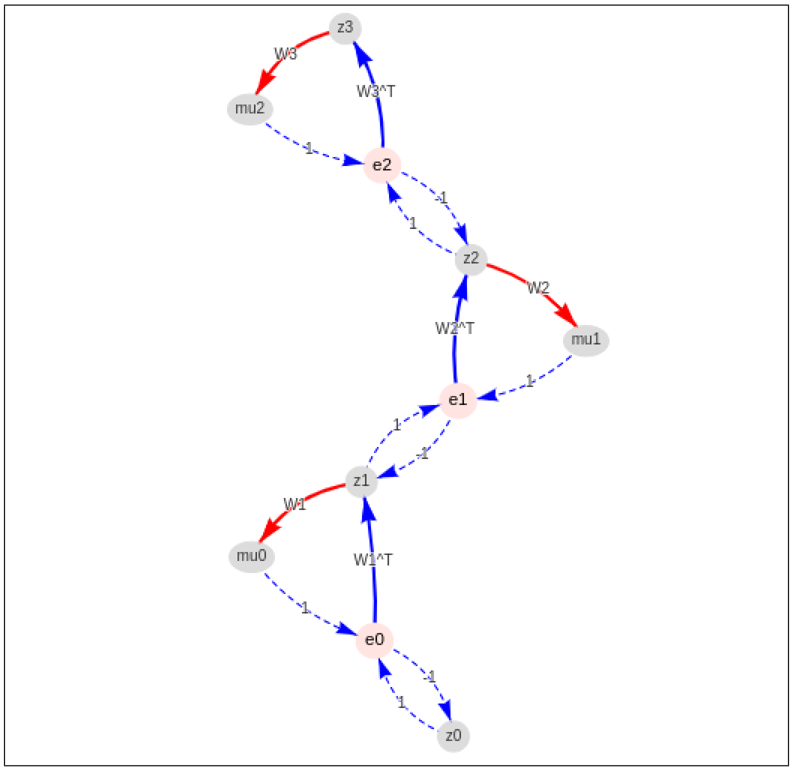
CHAPTER

TWELVE

GNCN-T1 (RAO & BALLARD, 1999)

This circuit implements the model proposed in (Rao & Ballard, 1999) [1]. Specifically, this model is **unsupervised** and can be used to process sensory pattern (row) vector(s) `x` to infer internal latent states. This class offers, beyond settling and update routines, a projection function by which ancestral sampling may be carried out given the underlying directed generative model formed by this NGC system.

The GNCN-t1 is graphically depicted by the following graph:



class `ngclearn.museum.gncn_t1.`**`GNCN_t1`**(*args*)

Structure for constructing the model proposed in:

Rao, Rajesh PN, and Dana H. Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects." Nature neuroscience 2.1 (1999): 79-87.

Note this model includes the Laplacian prior to induce some level of sparsity in the latent activities. This model, under the NGC computational framework, is referred to as the GNCN-t1/Rao, according to the naming convention in (Ororbia & Kifer 2022).

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-t1

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim - # of latent variables in layer z3 (top-most layer)
* z_dim - # of latent variables in layers z1 and z2
* x_dim - # of latent variables in layer z0 or sensory x
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* beta - latent state update factor
* leak - strength of the leak variable in the latent states
* lmbda - strength of the Laplacian prior applied over latent state activities
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1, z2, and z3
* out_fx - activation function for layer mu0 (prediction of z0) (Default: sigmoid)

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters `z_sample` – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Rao, Rajesh PN, and Dana H. Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects." Nature neuroscience 2.1 (1999): 79-87.

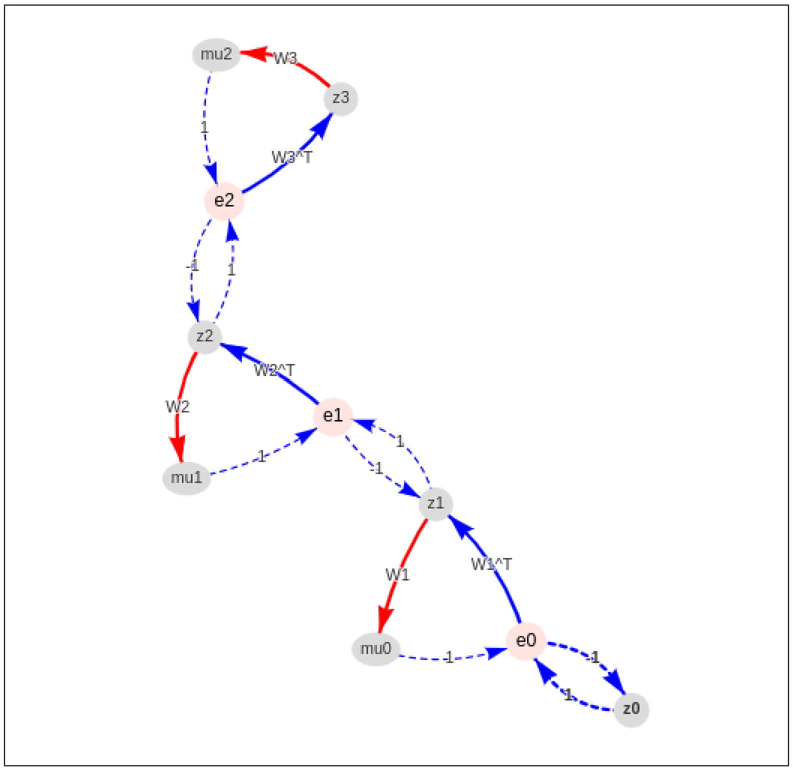
CHAPTER

THIRTEEN

GNCN-T1-SIGMA (FRISTON, 2008)

This circuit implements the model proposed in (Friston, 2008) [1]. Specifically, this model is **unsupervised** and can be used to process sensory pattern (row) vector(s) `x` to infer internal latent states. This class offers, beyond settling and update routines, a projection function by which ancestral sampling may be carried out given the underlying directed generative model formed by this NGC system.

The GNCN-t1-Sigma is graphically depicted by the following graph:



class `ngclearn.museum.gncn_t1_sigma.`**`GNCN_t1_Sigma`**(*args*)

Structure for constructing the model proposed in:

Friston, Karl. "Hierarchical models in the brain." PLoS Computational Biology 4.11 (2008): e1000211.

Note this model includes a Laplacian prior to induce some level of sparsity in the latent activities. This model, under the NGC computational framework, is referred to as the GNCN-t1-Sigma/Friston, according to the naming convention in (Ororbia & Kifer 2022).

Node Name Structure:

z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
e2 -> e2 * Sigma2; e1 -> e1 * Sigma1 // Precision weighting

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-t1-Sigma

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim: # of latent variables in layer z3 (top-most layer)
* z_dim: # of latent variables in layers z1 and z2
* x_dim: # of latent variables in layer z0 or sensory x
* seed: number to control determinism of weight initialization
* wght_sd: standard deviation of Gaussian initialization of weights
* beta: latent state update factor
* leak: strength of the leak variable in the latent states
* lmbda: strength of the Laplacian prior applied over latent state activities
* K: # of steps to take when conducting iterative inference/settling
* act_fx: activation function for layers z1, z2, and z3
* out_fx: activation function for layer mu0 (prediction of z0) (Default: sigmoid)

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters `z_sample` – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Friston, Karl. "Hierarchical models in the brain." PLoS Computational Biology 4.11 (2008): e1000211.

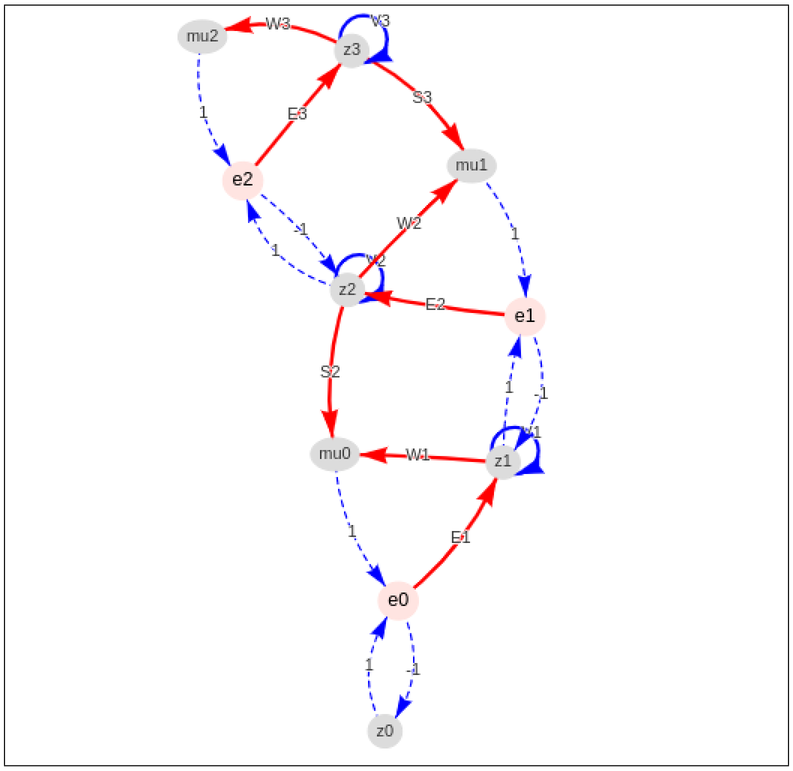
CHAPTER

FOURTEEN

GNCN-PDH (ORORBIA & KIFER, 2020/2022)

This circuit implements one of the models proposed in (Ororbia & Kifer, 2022) [1]. Specifically, this model is **unsupervised** and can be used to process sensory pattern (row) vector(s) `x` to infer internal latent states. This class offers, beyond settling and update routines, a projection function by which ancestral sampling may be carried out given the underlying directed generative model formed by this NGC system.

The GNCN-PDH is graphically depicted by the following graph:



class `ngclearn.museum.gncn_pdh.GNCN_PDH`(*args*)

Structure for constructing the model proposed in:

Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022).

This model, under the NGC computational framework, is referred to as the GNCN-t1-Sigma/Friston, according to the naming convention in (Ororbia & Kifer 2022).

Historical Note:

(The arXiv paper that preceded the publication above is shown below:)
Ororbia, Alexander, and Daniel Kifer. "The neural coding framework for
learning generative models." arXiv preprint arXiv:2012.03405 (2020).

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
z3 -(z3-mu1)-> mu1; z2 -(z2-mu0)-> mu0
e2 -> e2 * Sigma2; e1 -> e1 * Sigma1 // Precision weighting
z3 -> z3 * Lat3; z2 -> z2 * Lat2; z1 -> z1 * Lat1 // Lateral competition
e2 -(e2-z3)-> z3; e1 -(e1-z2)-> z2; e0 -(e0-z1)-> z1 // Error feedback

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-PDH

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim: # of latent variables in layer z3 (top-most layer)
* z_dim: # of latent variables in layers z1 and z2
* x_dim: # of latent variables in layer z0 or sensory x
* seed: number to control determinism of weight initialization
* wght_sd: standard deviation of Gaussian initialization of weights
* beta: latent state update factor
* leak: strength of the leak variable in the latent states
* K: # of steps to take when conducting iterative inference/settling
* act_fx: activation function for layers z1, z2, and z3
* out_fx: activation function for layer mu0 (prediction of z0) (Default: sigmoid)
* n_group: number of neurons w/in a competition group for z2 and z2 (sizes of z2 and z1 should be divisible by this number)
* n_top_group: number of neurons w/in a competition group for z3 (size of z3 should be divisible by this number)
* alpha_scale: the strength of self-excitation
* beta_scale: the strength of cross-inhibition

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters `z_sample` – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022).

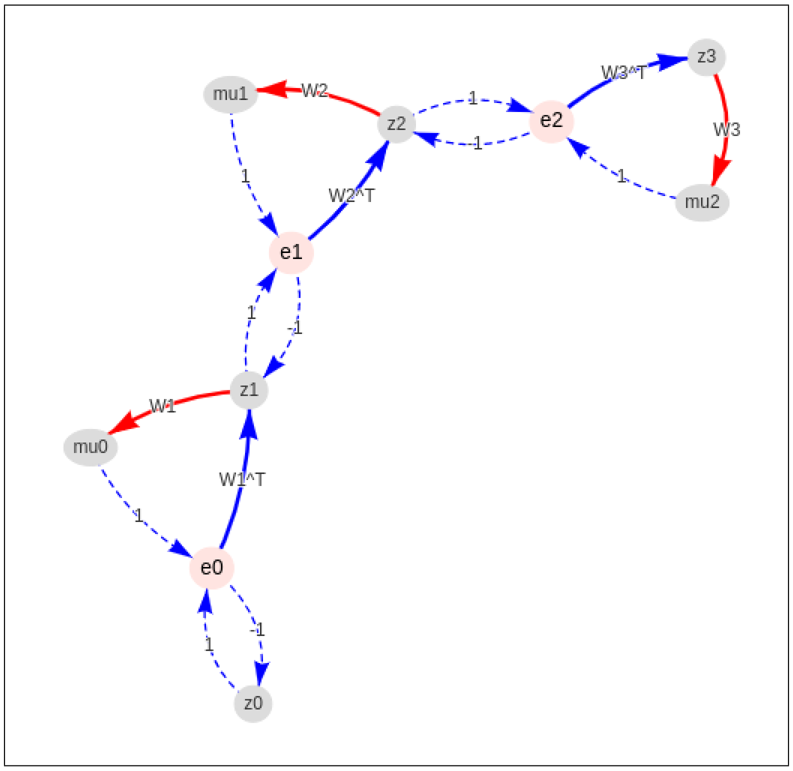
CHAPTER

FIFTEEN

GNCN-T1-FFM (WHITTINGTON & BOGACZ, 2017)

This circuit implements the model proposed in ((Whittington & Bogacz, 2017) [1]. Specifically, this model is **supervised** and can be used to process sensory pattern (row) vector(s) `x` to predict target (row) vector(s) `y`. This class offers, beyond settling and update routines, a prediction function by which ancestral projection is carried out to efficiently provide label distribution or regression vector outputs. Note that "FFM" denotes "feedforward mapping".

The GNCN-t1-FMM is graphically depicted by the following graph:



`class ngclearn.museum.gncn_t1_ffm.`**`GNCN_t1_FFM`**`(`*args*`)`

Structure for constructing the model proposed in:

Whittington, James CR, and Rafal Bogacz. "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity." Neural computation 29.5 (2017): 1229-1262.

This model, under the NGC computational framework, is referred to as the GNCN-t1-FFM, a slightly modified from of the naming convention in (Ororbia & Kifer 2022, Supplementary Material). "FFM" denotes feedforward mapping.

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
Note that z3 = x and z0 = y, yielding a classifier or regressor

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-t1-FFM

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* x_dim - # of latent variables in layer z3 or sensory input x
* z_dim - # of latent variables in layers z1 and z2
* y_dim - # of latent variables in layer z0 or output target y
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* beta - latent state update factor
* leak - strength of the leak variable in the latent states
* lmbda - strength of the Laplacian prior applied over latent state activities
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1, z2
* out_fx - activation function for layer mu0 (prediction of z0 or y) (Default: identity)

`predict`(*x*)

Predicts the target (either a probability distribution over labels, i.e., p(y|x), or a vector of regression targets) for a given *x*

Parameters `z_sample` – the input sample to project through the NGC graph

Returns y_sample (sample(s) of the underlying predictive model)

`settle`(*x*, *y*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to clamp top-most layer (z3) to
- `y` – target output activity, i.e., label or regression target
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns y_hat (predicted y)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Whittington, James CR, and Rafal Bogacz. "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity." Neural computation 29.5 (2017): 1229-1262.

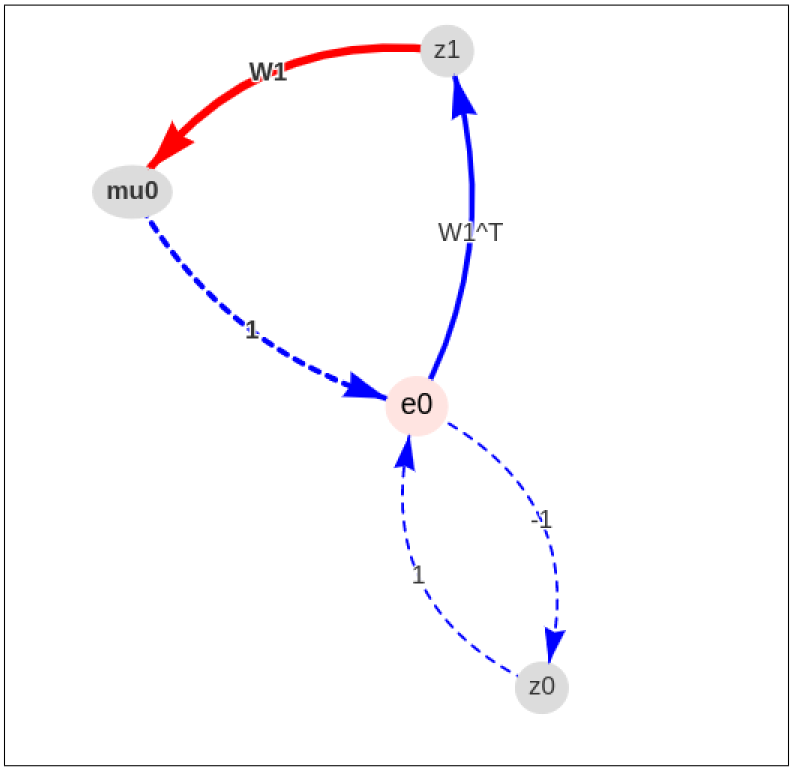
CHAPTER

SIXTEEN

GNCN-T1-SC (OLSHAUSEN & FIELD, 1996)

This circuit implements the sparse coding model proposed in (Olshausen & Field, 1996) [1]. Specifically, this model is **unsupervised** and can be used to process sensory pattern (row) vector(s) `x` to infer internal latent states. This class offers, beyond settling and update routines, a projection function by which ancestral sampling may be carried out given the underlying directed generative model formed by this NGC system.

The GNCN-t1-SC is graphically depicted by the following graph:



class `ngclearn.museum.gncn_t1_sc.`**`GNCN_t1_SC`**`(`*args*`)`

Structure for constructing the sparse coding model proposed in:

Olshausen, B., Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996).

Note this model imposes a factorial (Cauchy) prior to induce sparsity in the latent activities z1 (the latent codebook). Synapses initialized from a (fan-in) scaled uniform distribution. This model would be named, under the NGC computational framework naming convention (Ororbia & Kifer 2022), as the GNCN-t1/SC (SC = sparse coding) or GNCN-t1/Olshausen.

Node Name Structure:
p(z1) ; z1 -(z1-mu0)-> mu0 ;e0; z0
Cauchy prior applied for p(z1)

Note: You can also recover the model learned through ISTA by using, instead of a factorial prior over latents, a thresholding function such as the "soft_threshold". (Make sure you set "prior" to "none" in this case.) This results in the GNCN-t1/SC emulating a system similar to that proposed in:

Daubechies, Ingrid, Michel Defrise, and Christine De Mol. "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint." Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences 57.11 (2004): 1413-1457.

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-t1/SC

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_dim - # of latent variables in layers z1
* x_dim - # of latent variables in layer z0 or sensory x
* seed - number to control determinism of weight initialization
* beta - latent state update factor
* leak - strength of the leak variable in the latent states (Default = 0)
* prior - type of prior to use (Default = "cauchy")
* lmbda - strength of the prior applied over latent state activities (only if prior != "none")
* threshold - type of threshold to use (Default = "none")
* thr_lmbda - strength of the threshold applied over latent state activities (only if threshold != "none")
* n_group - must be > 0 if lat_type != None and s.t. (z_dim mod n_group) == 0
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1 (Default = identity)
* out_fx - activation function for layer mu0 (prediction of z0) (Default: identity)

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters `z_sample` – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`settle`(*x*, *K=- 1*, *cold_start=True*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `K` – number of steps to run iterative settling for
- `cold_start` – start settling process states from zero (Leave this to True)
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`calc_updates`(*avg_update=True*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Olshausen, B., Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996).

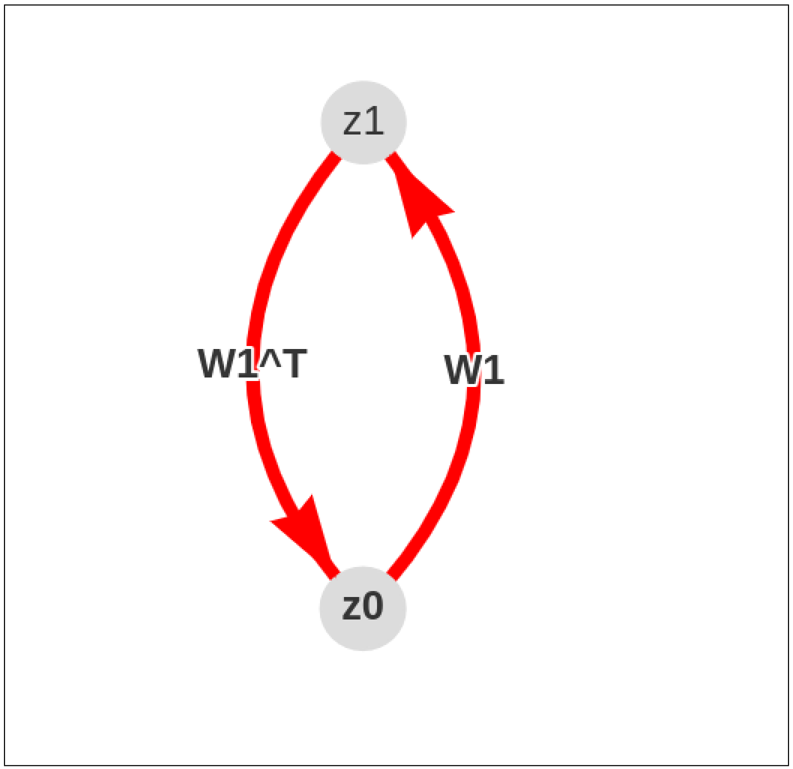
CHAPTER

SEVENTEEN

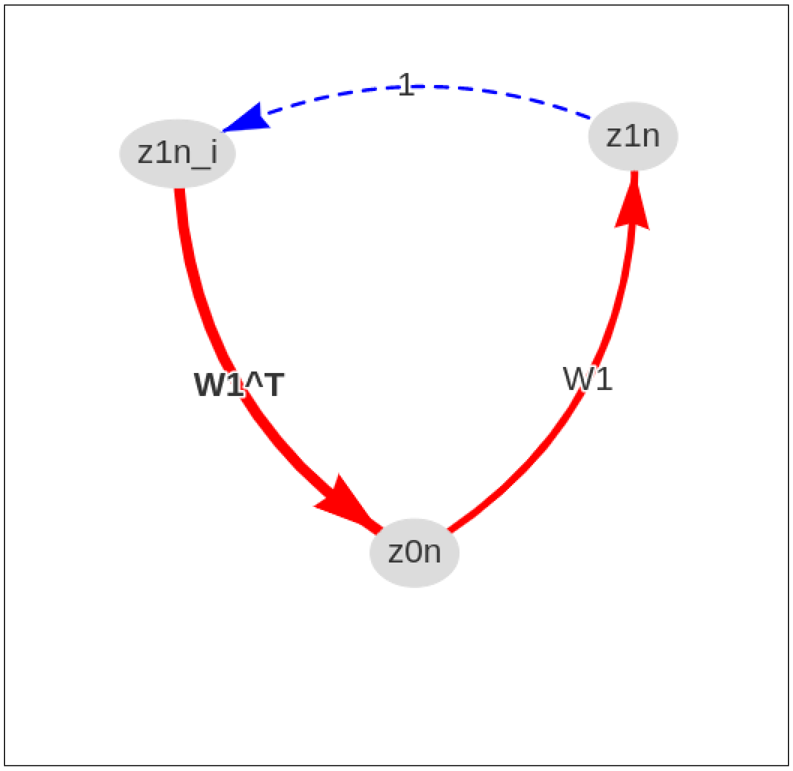
HARMONIUM (SMOLENSKY, 1986)

This circuit implements the Harmonium model proposed in (Smolensky, 1986) [1]. Specifically, this model is **unsupervised** and can be used to process sensory pattern (row) vector(s) `x` to infer internal latent states. This class offers, beyond settling and update routines through Contrastive Divergence (Hinton 1999) [2], a block Gibbs sampling function to generate a chain of synthesized patterns.

The Harmonium is technically defined by two NGC graphs. The first is the positive phase ("wake" phase) graph depicted graphically below:



while second is the negative phase ("sleep" phase) graph depicted graphically below:



class `ngclearn.museum.harmonium.`**`Harmonium`**(*args*)

Structure for constructing the Harmonium model proposed in:

Hinton, Geoffrey E. "Training products of experts by maximizing contrastive likelihood." Technical Report, Gatsby computational neuroscience unit (1999).

Node Name Structure:
z1 -(z1-z0)-> z0
z0 -(z0-z1)-> z1
Note: z1-z0 = (z0-z1)^T (transpose-tied synapses)

Another important reference for designing stable Harmoniums is here:

Hinton, Geoffrey E. "A practical guide to training restricted Boltzmann machines." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 599-619.

Note: if you set the *samp_fx* to the "identity", you force the Harmonium to to work as a mean-field Harmonium/Botlzmann machine

Parameters **`args`** – a Config dictionary containing necessary meta-parameters for the Harmonium

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_dim - # of latent variables in layer z1
* x_dim - # of latent variables in layer z0 (or sensory x)

* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* K - # of steps to take when conducting Contrastive Divergence
* act_fx - activation function for layer z1 (Default: sigmoid)
* out_fx - activation function for layer z0 (prediction of z0) (Default: sigmoid)
* samp_fx - sampling function for layer z1 (Default = bernoulli)

`sample`(*K*, *x_sample=None*, *batch_size=1*)

Samples the underlying harmonium to generate a chain of patterns from a block Gibbs sampling process.

Parameters

- **K** – number of steps to run the Gibbs sampler
- **x_sample** – inital condition for the sampler (Default = None), if None, this will generate an initial sample of size (batch_size, z1_dim) where z1_dim is the dimensionality of the latent state.
- **batch_size** – if x_sample is None, then this dictates how many samples in parallel to create per step of running the Gibbs sampler

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables.

Parameters

- **x** – sensory input to reconstruct/predict
- **calc_update** – if True, computes synaptic updates @ end of settling process for both NGC system and inference co-model (Default = True)

Returns x_hat (predicted x)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic updates (that follow order of pos_phase.theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Smolensky, P. "Information Processing in Dynamical Systems: Foundations of Harmony Theory." Parallel distributed processing: explorations in the microstructure of cognition 1 (1986). [2] Hinton, Geoffrey E. "Training products of experts by maximizing contrastive likelihood." Technical Report, Gatsby computational neuroscience unit (1999).

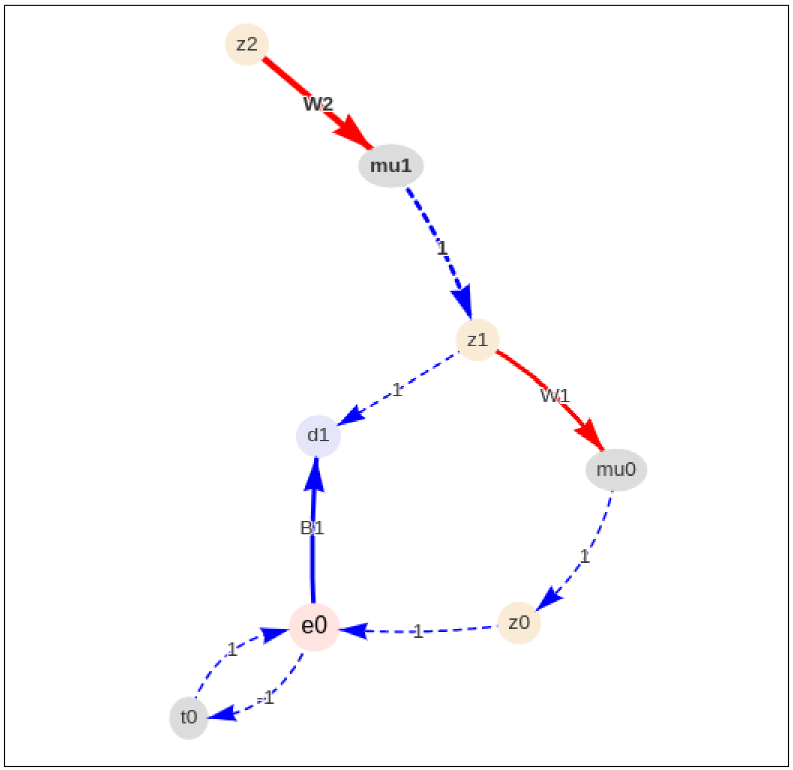
CHAPTER

EIGHTEEN

SNN-BA (SAMADI ET AL., 2017)

This circuit implements the spiking neural model of (Samadi et al., 2017) [1]. Specifically, this model is **supervised** and can be used to process sensory pattern (row) vector(s) `x` to predict target (row) vector(s) `y`. This class offers, beyond settling and update routines, a prediction function by which ancestral projection is carried out to efficiently provide label distribution or regression vector outputs. Note that "SNN" denotes "spiking neural network" and "BA" stands for "broadcast alignment". This class model it does not feature a separate `calc_updates()` method like other models since its `settle()` routine adjusts synaptic efficacies dynamically (if configured to do so).

The SNN-BA is graphically depicted by the following graph:



class `ngclearn.museum.snn_ba.`**`SNN_BA`**(*args*)

A spiking neural network (SNN) classifier that adapts its synaptic cables via broadcast alignment. Specifically, this model is a generalization of the one proposed in:

Samadi, Arash, Timothy P. Lillicrap, and Douglas B. Tweed. "Deep learning with dynamic spiking neurons and fixed feedback weights." Neural computation 29.3 (2017): 578-602.

This model encodes its real-valued inputs as Poisson spike trains with spikes emitted at a rate of approximately 63.75 Hz. The internal nodes and output nodes operate under the leaky integrate-and-fire spike response model and operate with a relative refractory rate of 1.0 ms. The integration time constant for this model has been set to 0.25 ms.

Node Name Structure:
z2 -(z2-mu1)-> mu1 ; z1 -(z1-mu0-)-> mu0 ;e0; z0
e0 -> d1 and z1 -> d1, where d1 is a teaching signal for z1
Note that z2 = x and z0 = y, yielding a classifier

Parameters `args` – a Config dictionary containing necessary meta-parameters for the SNN-BA

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_dim - # of latent variables in layers z1
* x_dim - # of latent variables in layer z2 or sensory x
* y_dim - # of variables in layer z0 or target y
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights (optional)
* T - # of time steps to take when conducting iterative settling (if not online)

`predict`(*x*)

Predicts the target for a given *x*. Specifically, this function will return spike counts, one per class in *y* – taking the argmax of these counts will yield the model's predicted label.

Parameters `z_sample` – the input sample to project through the NGC graph

Returns y_sample (spike counts from the underlying predictive model)

`settle`(*x*, *y=None*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables, specifically simulating the dynamics of the spiking neurons internal to this SNN model. Note that this functions returns two outputs – the first is a count matrix (each row is a sample in mini-batch) and each column represents the count for one class in y, and the second is an approximate probability distribution computed as a softmax over an average across the electrical currents produced at each step of simulation.

Parameters

- `x` – sensory input to clamp top-most layer (z2) to
- `y` – target output activity, i.e., label target
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns

y_count (spike counts per class in y), y_hat (approximate probability distribution for y)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

References: [1] Samadi, Arash, Timothy P. Lillicrap, and Douglas B. Tweed. "Deep learning with dynamic spiking neurons and fixed feedback weights." Neural computation 29.3 (2017): 578-602.

CHAPTER

NINETEEN

NODE

A Node represents one of the fundamental building blocks of an NGC system. These particular objects are meant to perform, per simulated time step, a calculation of output activity values given an internal arrangement of compartments (or sources where signals from other Node(s) are to be deposited).

19.1 Node Model

The `Node` class serves as a root class for the node building block objects of an NGC system/graph. This is a core modeling component of general NGC computational systems. Node sub-classes within ngc-learn inherit from this base class.

`class ngclearn.engine.nodes.node.`**`Node`**`(`*node_type*, *name*, *dim*`)`

Base node element (class from which other node types inherit basic properties from)

Parameters

- **`node_type`** – the string concretely denoting this node's type
- **`name`** – str name of this node
- **`dim`** – number of neurons this node will contain

`wire_to``(`*dest_node*, *src_comp*, *dest_comp*, *cable_kernel=None*, *mirror_path_kernel=None*, *name=None*, *short_name=None*`)`

A wiring function that connects this node to another external node via a cable (or synaptic bundle)

Parameters

- **`dest_node`** – destination node (a Node object) to wire this node to
- **`src_comp`** – name of the compartment inside this node to transmit a signal from (to destination node)
- **`dest_comp`** – name of the compartment inside the destination node to transmit a signal to
- **`cable_kernel`** – Dict defining how to initialize the cable that will connect this node to the destination node. The expected keys and corresponding value types are specified below:

  'type' type of cable to be created. If "dense" is specified, a DCable (dense cable/bundle/matrix of synapses) will be used to transmit/transform information along.

  'init_kernels' a Dict specifying how parameters w/in the learnable parts of the cable are to randomly initialized

  'seed' integer seed to deterministically control initialization of synapses in a DCable

Note either cable_kernel, mirror_path_kernel MUST be set to something that is not None

- `mirror_path_kernel` – 2-Tuple that allows a currently existing cable to be re-used as a transformation. The value types inside each slot of the tuple are specified below:

  cable_to_reuse (Tuple[0]) target cable (usually an existing DCable object) to shallow copy and mirror

  mirror_type (Tuple[1]) how should the cable be mirrored? If "symm_tied" is specified, then the transpose of this cable will be used to transmit information from this node to a destination node, if "anti_symm_tied" is specified, the negative transpose of this cable will be used, and if "tied" is specified, then this cable will be used exactly in the same way it was used in its source cable.

  Note either cable_kernel, mirror_path_kernel MUST be set to something that is not None

- `name` – the string name to be assigned to the generated cable (Default = None)

  Note setting this to None will trigger the created cable to auto-name itself

`inject`(*data*)

Injects an externally provided named value (a vector/matrix) to the desired compartment within this node.

Parameters `data` – 2-Tuple containing a named external signal to clamp

compartment_name (Tuple[0]) the (str) name of the compartment to clamp this data signal to.

signal (Tuple[1]) the data signal block to clamp to the desired compartment name

`clamp`(*data*, *is_persistent=True*)

Clamps an externally provided named value (a vector/matrix) to the desired compartment within this node.

Parameters

- `data` – 2-Tuple containing a named external signal to clamp

  compartment_name (Tuple[0]) the (str) name of the compartment to clamp this data signal to.

  signal (Tuple[1]) the data signal block to clamp to the desired compartment name

- `is_persistent` – if True, prevents this node from overriding the clamped data over time (Default = True)

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- `injection_table` –
- `skip_core_calc` – skips the core components of this node's calculation (Default = False)

`calc_update`(*update_radius=- 1.0*)

Calculates the updates to local internal synaptic parameters related to this specific node given current relevant values (such as node-level precision matrices).

Parameters **`update_radius`** – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

`clear`()

Wipes/clears values of each compartment in this node (and sets .is_clamped = False).

`extract`(*comp_name*)

Extracts the data signal value that is currently stored inside of a target compartment

Parameters **`comp_name`** – the name of the compartment in this node to extract data from

`extract_params`()

`deep_store_state`()

Performs a deep copy of all compartment statistics.

Returns Dict containing a deep copy of each named compartment of this node

19.2 SNode Model

The `SNode` class extends from the base `Node` class, and represents a (rate-coded) state node that follows a certain set of settling dynamics. In conjunction with the corresponding `ENode` and `FNode` classes, this serves as the core modeling component of a higher-level `NGCGraph` class used in simulation.

class `ngclearn.engine.nodes.snode.`**`SNode`**(*name*, *dim*, *beta=1.0*, *leak=0.0*, *zeta=1.0*, *act_fx='identity'*, *batch_size=1*, *integrate_kernel=None*, *prior_kernel=None*, *threshold_kernel=None*, *trace_kernel=None*, *samp_fx='identity'*)

Implements a (rate-coded) state node that follows NGC settling dynamics according to:

d.z/d.t = -z * leak + dz + prior(z), where dz = dz_td + dz_bu * phi'(z)

where:

dz - aggregated input signals from other nodes/locations

leak - controls strength of leak variable/decay

prior(z) - distributional prior placed over z (such as a kurtotic prior)

Note that the above is used to adjust neural activity values via an integator inside a node. For example, if the standard/default Euler integrator is used then the neurons inside this node are adjusted per step as follows:

z <- z * zeta + d.z/d.t * beta

where:

beta - strength of update to node state z

zeta - controls the strength of recurrent carry-over, if set to 0 no carry-over is used (stateless)

Compartments:

* dz_td - the top-down pressure compartment (deposited signals summed)
* dz_bu - the bottom-up pressure compartment, potentially weighted by phi'(x)) (deposited signals summed)
* z - the state neural activities
* phi(z) - the post-activation of the state activities
* S(z) - the sampled state of phi(z) (Default = identity or f(phi(z)) = phi(z))

* mask - a binary mask to be applied to the neural activities

Parameters

- `name` – the name/label of this node
- `dim` – number of neurons this node will contain/model
- `beta` – strength of update to adjust neurons at each simulation step (Default = 1)
- `leak` – strength of the leak applied to each neuron (Default = 0)
- `zeta` – effect of recurrent/stateful carry-over (Defaul = 1)
- `act_fx` – activation function – phi(v) – to apply to neural activities

  Note if using either "kwta" or "bkwta", please input how many winners should win the competiton, i.e., use "kwta(N)" or "bkwta(N)" where N is an integer > 0.

- `batch_size` – batch-size this node should assume (for use with static graph optimization)
- `integrate_kernel` – Dict defining the neural state integration process type. The expected keys and corresponding value types are specified below:

  'integrate_type' type integration method to apply to neural activity over time. If "euler" is specified, Euler integration will be used (future ngc-learn versions will support "midpoint"/other methods).

  'use_dfx' a boolean that decides if phi'(v) (activation derivative) is used in the integration process/update.

  Note specifying None will automatically set this node to use Euler integration w/ use_dfx=False

- `prior_kernel` – Dict defining the type of prior function to apply over neural activities. The expected keys and corresponding value types are specified below:

  'prior_type' type of (centered) distribution to use as a prior over neural activities. If "laplace" is specified, a Laplacian distribution is used, if "cauchy" is specified, a Cauchy distribution will be used, if "gaussian" is specified, a Gaussian distribution will be used, and if "exp" is specified, the exponential distribution will be used.

  'lambda' the scale factor controlling the strength of the prior applied to neural activities.

  Note specifying None will result in no prior distribution being applied

- `threshold_kernel` – Dict defining the type of threshold function to apply over neural activities. The expected keys and corresponding value types are specified below:

  'threshold_type' type of (centered) distribution to use as a prior over neural activities. If "soft_threshold" is specified, a soft thresholding function is used, and if "cauchy_threshold" is specified, a cauchy thresholding function is used,

  'thr_lambda' the scale factor controlling the strength of the threshold applied to neural activities.

  Note specifying None will result in no threshold function being applied

- `trace_kernel` – <unused> (Default = None)

- **samp_fx** – the sampling/stochastic activation function – S(v) – to apply to neural activities (Default = identity)

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **injection_table** –
- **skip_core_calc** – skips the core components of this node's calculation (Default = False)

`clear`()

Wipes/clears values of each compartment in this node (and sets .is_clamped = False).

19.3 ENode Model

The `ENode` class extends from the base `Node` class, and represents a (rate-coded) error node simplified to its fixed-point form. In conjunction with the corresponding `SNode` and `FNode` classes, this serves as the core modeling component of a higher-level `NGCGraph` class used in simulation.

class `ngclearn.engine.nodes.enode.ENode`(*name*, *dim*, *error_type='mse'*, *act_fx='identity'*, *batch_size=1*, *precis_kernel=None*, *constraint_kernel=None*, *ex_scale=1.0*)

Implements a (rate-coded) error node simplified to its fixed-point form:

e = target - mu // in the case of squared error (Gaussian error units)

e = signum(target - mu) // in the case of absolute error (Laplace error units)

where:

target - a desired target activity value (target = pred_targ)

mu - an external prediction signal of the target activity value (mu = pred_mu)

Compartments:

* pred_mu - prediction signals (deposited signals summed)

* pred_targ - target signals (deposited signals summed)

* z - the error neural activities, set as z = e

* phi(z) - the post-activation of the error activities in z

* L - the local loss represented by the error activities

* avg_scalar - multiplies L and z by (1/avg_scalar)

Parameters

- **name** – the name/label of this node
- **dim** – number of neurons this node will contain/model
- **error_type** – type of distance/error measured by this error node. Setting this to "mse" will set up squared-error neuronal units (derived from L = 0.5 * (Sum_j (target - mu)^2_j)), and "mae" will set up mean absolute error neuronal units (derived from L = Sum_j |target - mu|).
- **act_fx** – activation function – phi(v) – to apply to error activities (Default = "identity")

- `batch_size` – batch-size this node should assume (for use with static graph optimization)
- `precis_kernel` – 2-Tuple defining the initialization of the precision weighting synapses that will modulate the error neural activities. For example, an argument could be: (“uniform”, 0.01) The value types inside each slot of the tuple are specified below:

  init_scheme (Tuple[0]) initialization scheme, e.g., “uniform”, “gaussian”.

  init_scale (Tuple[1]) scalar factor controlling the scale/magnitude of initialization distribution, e.g., 0.01.

  Note specifying None will result in precision weighting being applied to the error neurons. Understand that care should be taken w/ respect to this particular argument as precision synapses involve an approximate inversion throughout simulation steps
- `constraint_kernel` – Dict defining the constraint type to be applied to the learnable parameters of this node. The expected keys and corresponding value types are specified below:

  ’clip_type’ type of clipping constraint to be applied to learnable parameters/synapses. If “norm_clip” is specified, then norm-clipping will be applied (with a check if the norm exceeds “clip_mag”), and if “forced_norm_clip” then norm-clipping will be applied regardless each time apply_constraint() is called.

  ’clip_mag’ the magnitude of the worse-case bounds of the clip to apply/enforce.

  ’clip_axis’ the axis along which the clipping is to be applied (to each matrix).

  Note specifying None will mean no constraints are applied to this node’s parameters
- `ex_scale` – a scale factor to amplify error neuron signals (Default = 1)

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters
- `injection_table` –
- `skip_core_calc` – skips the core components of this node’s calculation (Default = False)

`calc_update`(*update_radius=- 1.0*)

Calculates the updates to local internal synaptic parameters related to this specific node given current relevant values (such as node-level precision matrices).

Parameters `update_radius` – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

`compute_precision`(*rebuild_cov=True*)

Co-function that pre-computes the precision matrices for this NGC node. NGC uses the Cholesky-decomposition form of precision (Sigma)^{-1}

Parameters `rebuild_cov` – rebuild the underlying covariance matrix after re-computing precision (Default = True)

`clear`()

Wipes/clears values of each compartment in this node (and sets .is_clamped = False).

19.4 FNode Model

The `FNode` class extends from the base `Node` class, and represents a stateless node that simply aggregates (via summation) its received inputs. In conjunction with the corresponding `SNode` and `ENode` classes, this serves as the core modeling component of a higher-level `NGCGraph` class used in simulation.

class `ngclearn.engine.nodes.fnode.`**`FNode`**(*name*, *dim*, *act_fx='identity'*, *batch_size=1*)

Implements a feedforward (stateless) transmission node:

z = dz

where:

dz - aggregated input signals from other nodes/locations

Compartments:

* dz - incoming pressures/signals (deposited signals summed)
* z - the state values/neural activities, set as: z = dz
* phi(z) - the post-activation of the neural activities

Parameters

- **`name`** – the name/label of this node
- **`dim`** – number of neurons this node will contain/model
- **`act_fx`** – activation function – phi(v) – to apply to neural activities
- **`batch_size`** – batch-size this node should assume (for use with static graph optimization)

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **`injection_table`** –
- **`skip_core_calc`** – skips the core components of this node's calculation (Default = False)

CHAPTER

TWENTY

CABLE

A Cable represents one of the fundamental building blocks of an NGC system. These particular objects are meant to serve as the connectors between Node(s), passing along or transforming signals from the source point (a compartment, or receiving area, within a particular node) to a destination point (another compartment in a different node) and transforming such signals through synaptic parameters.

20.1 Cable Model

The `Cable` class serves as a root class for the wire building block objects of an NGC system/graph. This is a core modeling component of general NGC computational systems. Cable sub-classes within ngc-learn inherit from this base class.

`class ngclearn.engine.cables.cable.Cable`(*cable_type*, *inp*, *out*, *name=None*, *seed=69*)

Base cable element (class from which other cable types inherit basic properties from)

Parameters

- **`cable_type`** – the string concretely denoting this cable's type
- **`inp`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- **`out`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

  input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- **`name`** – the string name of this cable (Default = None which creates an auto-name)
- **`seed`** – integer seed to control determinism of any underlying synapses associated with this cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

`set_update_rule`(*preact=None*, *postact=None*, *update_rule=None*, *gamma=1.0*, *use_mod_factor=False*, *param=None*, *decay_kernel=None*)

Sets the synaptic adjustment rule for this cable (currently a 2-factor local synaptic Hebbian update rule).

Parameters

- `preact` – 2-Tuple defining the pre-activity/source node of which the first factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

  preact_node (Tuple[0]) the physical node that offers a pre-activity signal for the first factor of the synaptic/cable update

  preact_compartment (Tuple[1]) the component in the preact_node to extract the necessary signal to compute the first factor the synaptic/cable update

- `postact` – 2-Tuple defining the post-activity/source node of which the second factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

  postact_node (Tuple[0]) the physical node that offers a post-activity signal for the second factor of the synaptic/cable update

  postact_compartment (Tuple[1]) the component in the postact_node to extract the necessary signal to compute the second factor the synaptic/cable update

- `update_rule` – a specific update rule to use with the parameters of this cable
- `gamma` – scaling factor for the synaptic update
- `use_mod_factor` – if True, triggers the modulatory matrix weighting factor to be applied to the resultant synaptic update

  Note This is un-tested/not fully integrated

- `param` – a list of strings, each containing named parameters that are to be learned w/in this cable
- `decay_kernel` – 2-Tuple defining the type of weight decay to be applied to the synapses. The value types inside each slot of the tuple are specified below:

  decay_type (Tuple[0]) string indicating which type of weight decay to use, "l2" will trigger L2-penalty decay, while "l1" will trigger L1-penalty decay

  decay_coefficient (Tuple[1]) scalar/float to control magnitude of decay applied to computed local updates

`calc_update`()

Calculates the updates to the internal synapses that compose this cable given this cable's pre-configured synaptic update rule.

Parameters `clip_kernel` – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

20.2 DCable Model

The `DCable` class extends from the base `Cable` class and represents a dense transform of signals from one nodal point to another. Signals that travel across it through a set of synaptic parameters (and potentially a base-rate/bias shift parameter). In conjunction with the corresponding `SCable` class, this serves as the core modeling component of a higher-level `NGCGraph` class used in simulation.

`class ngclearn.engine.cables.dcable.DCable`(*inp*, *out*, *init_kernels=None*, *shared_param_path=None*, *clip_kernel=None*, *constraint_kernel=None*, *seed=69*, *name=None*)

A dense cable that transforms signals that travel across via a bundle of synapses. (In other words, a linear projection followed by an optional base-rate/bias shift.)

Note: a dense cable only contains two possible learnable parameters, "A" and "b" each with only two terms for their local Hebbian updates.

Parameters

- **`inp`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- **`out`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

  input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- **`w_kernel`** – an N-Tuple defining type of scheme to randomly initialize weights.

  scheme (Tuple[0]) triggers the type of initalization scheme, for example, "gaussian" will apply an elementwise Gaussian initialization. (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their string codes that can be used.)

  scheme_arg1 (Tuple[1]) first argument to control the initialization (for many schemes, setting this value to 1.0 or even omitting it is acceptable given that this parameter is ignored, for example, in "unif_scale", the second argument would be ignored.) (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their extra arguments.)

  scheme_arg2 (Tuple[2]) second argument to control the initialization – this is generally only necessary to set in the case of lateral competition initialization schemes, such as in the case of "lkwta" which requires a 3-Tuple specified as follows: ("lkwta",alpha_scale,beta_scale) where alpha_scale controls the strength of self-excitation and beta_scale controls the strength of the cross-unit inhibition.

- **`b_kernel`** – 2-Tuple defining type of scheme to randomly initialize weights.

scheme (Tuple[0]) triggers the type of initalization scheme, for example, "gaussian" will apply an elementwise Gaussian initialization. (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their string codes that can be used.)

scheme_arg1 (Tuple[1]) first argument to control the initialization (for many schemes, setting this value to 1.0 or even omitting it is acceptable given that this parameter is ignored, for example, in "unif_scale", the second argument would be ignored.) (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their extra arguments.)

- **shared_param_path** –
- **clip_kernel** – 3-Tuple defining type of clipping to apply to calculated synaptic adjustments.

  clip_type (Tuple[0]) type of clipping constraint to apply. If "hard_clip" is set, then a hard-clipping routine is applied (ignoring "clip_axis") while "norm_clip" clips by checking if the norm exceeds "clip_value" along "clip_axis". Note that "hard_clip" will also be applied to biases (while "clip_norm" is not).

  clip_value (Tuple[1]) the magnitude of the worse-case bounds of the clip to apply/enforce.

  clip_axis (Tuple[2]) the axis along which the clipping is to be applied (to each matrix).

  Note specifying None will mean no clipping is applied to this cable's calculated updates

- **constraint_kernel** – Dict defining the constraint type to be applied to the learnable parameters of this cable. The expected keys and corresponding value types are specified below:

  'clip_type' type of clipping constraint to be applied to learnable parameters/synapses. If "norm_clip" is specified, then norm-clipping will be applied (with a check if the norm exceeds "clip_mag"), and if "forced_norm_clip" then norm-clipping will be applied regardless each time apply_constraint() is called.

  'clip_mag' the magnitude of the worse-case bounds of the clip to apply/enforce.

  'clip_axis' the axis along which the clipping is to be applied (to each matrix).

  Note specifying None will mean no constraints are applied to this cable's parameters

- **name** – the string name of this cable (Default = None which creates an auto-name)
- **seed** – integer seed to control determinism of any underlying synapses associated with this cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

`calc_update()`

Calculates the updates to the internal synapses that compose this cable given this cable's pre-configured synaptic update rule.

Parameters **`clip_kernel`** – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

20.3 SCable Model

The `SCable` class extends from the base `Cable` class and represents a simple carry-over of signals from one nodal point to another. Signals that travel across it can either be carried directly (an identity transform) or multiplied by a scalar amplification coefficient. In conjunction with the corresponding `DCable` class, this serves as the core modeling component of a higher-level `NGCGraph` class used in simulation.

`class ngclearn.engine.cables.scable.SCable(`*inp*, *out*, *coeff=1.0*, *name=None*, *seed=69*`)`

A simple cable that, at most, applies a scalar amplification of signals that travel across it. (Otherwise, this cable works like an identity carry-over.)

Parameters

- **`inp`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- **`out`** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

  input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- **`coeff`** – a scalar float to control any signal scaling associated with this cable
- **`name`** – the string name of this cable (Default = None which creates an auto-name)
- **`seed`** – integer seed to control determinism of any underlying synapses associated with this cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

CHAPTER

TWENTYONE

NGCGRAPH

An NGCGraph represents one of the core structural components of an NGC system. This particular object is what Node(s) and Cable(s) are ultimately embedded/integrated into in order to simulate a full NGC process (key functions include the primary settling process and synaptic update calculation routine). Furthermore, the NGCGraph contains several tool functions to facilitate analysis of the system evolved over time.

21.1 NGC Graph

The `NGCGraph` class serves as a core building block for forming a complete NGC computational processing system.

`class ngclearn.engine.ngc_graph.NGCGraph(`*K=5*, *name='ncn'*, *batch_size=1*`)`

Implements the full model structure/graph for an NGC system composed of nodes and cables. Note that when instantiating this object, it is important to call .compile(), like so:

graph = NGCGraph(. . .)
info = graph.compile()

Parameters

- **`K`** – number of iterative inference/settling steps to simulate
- **`name`** – (optional) the name of this projection graph (Default="ncn")
- **`batch_size`** – fixed batch-size that the underlying compiled static graph system should assume (Note that you can set this also as an argument to .compile())

  Note if "use_graph_optim" is set to False, then this argument is not meaningful as the system will work with variable-length batches

`set_cycle``(`*nodes*, *param_order=None*`)`

Set an execution cycle in this graph

Parameters **`nodes`** – an ordered list of Node(s) to create an execution cycle for

`compile``(`*use_graph_optim=True*, *batch_size=- 1*`)`

Executes a global "compile" of this simulation object to ensure internal system coherence. (Only call this function after the constructor has been set).

Parameters

- **`use_graph_optim`** – if True, this simulation will use static graph acceleration (Default = True)

- `batch_size` – if > 0, will set the integer global batch_size of this simulation object (otherwise, self.batch_size will be used)

Returns a dictionary containing post-compilation information about this simulation object

`clone_state()`

Clones the entire state of this graph (in terms of signals/tensors) and stores each node's state dictionary in global has map

Returns a Dict (hash table) containing string names that map to physical Node objects

`set_to_state`(*state_map*)

Set every state of every node in this graph to the values contained in the global Dict (hash table) "state_map"

Parameters `state_map` – a Dict (hash table) containing string names that map to physical Node objects

`extract`(*node_name*, *node_var_name*)

Extract a particular signal from a particular node embedded in this graph

Parameters

- `node_name` – name of the node from the NGC graph to examine
- `node_var_name` – compartment name w/in Node to extract signal from

Returns an extracted signal (vector/matrix) OR None if either the node does not exist or the entire system has not been simulated (meaning that no node dynamics have been run yet)

`getNode`(*node_name*)

Extract a particular node from this graph

Parameters `node_name` – name of the node from the NGC graph to examine

Returns the desired Node (object) or None if the node does not exist

`clamp`(*clamp_targets*)

Clamps an externally provided named value (a vector/matrix) to the desired compartment within a particular Node of this NGC graph. Note that clamping means this value typically means the value clamped on will persist (it will NOT evolve according to the injected node's dynamics over simulation steps, unless is_persistent = True).

Parameters `clamp_targets` – 3-Tuple containing a named external signal to clamp

node_name (Tuple[0]) the (str) name of the node to clamp a data signal to.

compartment_name (Tuple[1]) the (str) name of the node's compartment to clamp this data signal to.

signal (Tuple[2]) the data signal block to clamp to the desired compartment name

`inject`(*injection_targets*)

Injects an externally provided named value (a vector/matrix) to the desired compartment within a particular Node of this NGC graph. Note that injection means this value does not persist (it will evolve according to the injected node's dynamics over simulation steps).

Parameters

- `injection_targets` – 3-Tuple containing a named external signal to clamp

  node_name (Tuple[0]) the (str) name of the node to clamp a data signal to.

  compartment_name (Tuple[1]) the (str) name of the compartment to clamp this data signal to.

signal (Tuple[2]) the data signal block to clamp to the desired compartment name

- `is_persistent` – if True, clamped data value will persist throughout simulation (Default = True)

`settle`(*clamped_vars=None*, *readout_vars=None*, *init_vars=None*, *cold_start=True*, *K=- 1*, *debug=False*, *masked_vars=None*, *calc_delta=True*)

Execute this NGC graph's iterative inference using the execution pathway(s) defined at construction/initialization.

Parameters

- `clamped_vars` – list of 3-tuple strings containing named Nodes, their compartments, and values to (persistently) clamp on. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]
- `readout_vars` – list of 2-tuple strings containing Nodes and their compartments to read from (in this function's output). Note that this list takes the form: [(node1_name, node1_compartment), node2_name, node2_compartment),. . .]
- `init_vars` – list of 3-tuple strings containing named Nodes, their compartments, and values to initialize each Node from. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]
- `cold_start` – initialize all non-clamped/initialized Nodes (i.e., their compartments contain None) to zero-vector starting points/resting states
- `K` – number simulation steps to run (Default = -1), if <= 0, then self.K will be used instead
- `debug` – <UNUSED>
- `masked_vars` – list of 4-tuple that instruct which nodes/compartments/masks/clamped values to apply. This list is used to trigger auto-associative recalls from this NGC graph. Note that this list takes the form: [(node1_name, node1_compartment, mask, value), node2_name, node2_compartment, mask, value),. . .]
- `calc_delta` – compute the list of synaptic updates for each learnable parameter within .theta? (Default = True)

Returns

readouts, delta; where "readouts" is a 3-tuple list of the form [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .], and "delta" is a list of synaptic adjustment matrices (in the same order as .theta)

`calc_updates`(*debug_map=None*)

Calculates the updates to synaptic weight matrices along each learnable wire within this graph via a generalized Hebbian learning rule.

Parameters `debug_map` – (Default = None), a Dict to place named signals inside (for debugging)

`apply_constraints`()

Apply any constraints to the signals embedded in this graph. This function will execute any of the following pre-configured constraints:

1) compute new precision matrices (if applicable)

2) project weights to adhere to vector norm constraints

clear()

Clears/deletes any persistent signals currently embedded w/in this graph's Nodes

CHAPTER

TWENTYTWO

PROJECTIONGRAPH

A ProjectionGraph represents one of the core structural components of an NGC system. To use a projection graph, Node(s) and Cable(s) must be embedded/integrated into in order to simulate an ancestral projection/sampling process. Note that ProjectionGraph is only useful if an NGCGraph has been created, given that a projection graph is meant to offer non-trainable functionality, particularly fast inference, to an NGC computational system.

22.1 Projection Graph

The `ProjectionGraph` class serves as a core building block for forming a complete NGC computational processing system (particularly with respect to external processes such as projection/sampling).

`class ngclearn.engine.proj_graph.`**`ProjectionGraph`**`(`*name='sampler'*`)`

Implements a projection graph – useful for conducting ancestral sampling of a directed generative model or ancestral projection of a clamped graph. Note that when instantiating this object, it is important to call .compile(), like so:

graph = ProjectionGraph(. . .)
info = graph.compile()

Parameters **`name`** – the name of this projection graph

`set_cycle``(`*nodes*`)`

Set execution cycle for this graph

Parameters **`nodes`** – an ordered list of Node(s) to create an execution cycle for

`extract``(`*node_name*, *node_var_name*`)`

Extract a particular signal from a particular node embedded in this graph

Parameters

- **`node_name`** – name of the node from the NGC graph to examine
- **`node_var_name`** – compartment name w/in Node to extract signal from

Returns an extracted signal (vector/matrix) OR None if node does not exist

`getNode``(`*node_name*`)`

Extract a particular node from this graph

Parameters **`node_name`** – name of the node from the NGC graph to examine

Returns the desired Node (object)

`project`(*clamped_vars=None*, *readout_vars=None*)

Project signals through the execution pathway(s) defined by this graph

Parameters

- **`clamped_vars`** – list of 2-tuples containing named Nodes that will be clamped with particular values. Note that this list takes the form: [(node1_name, node_value1), node2_name, node_value2),. . .]
- **`readout_vars`** – list of 2-tuple strings containing named Nodes and their compartments to read signals from. Note that this list takes the form: [(node1_name, node1_compartment), node2_name, node2_compartment),. . .]

Returns

readout values - a list of 3-tuples named signals corresponding to the ones in "readout_vars". Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]

`clear`()

Clears/deletes any persistent signals currently embedded w/in this graph's Nodes

CHAPTER

TWENTYTHREE

NGCLEARN

23.1 ngclearn package

23.1.1 Subpackages

ngclearn.density package

Submodules

ngclearn.density.gmm module

class `ngclearn.density.gmm.`**`GMM`**(*k*, *max_iter=5*, *assume_diag_cov=False*, *init_kmeans=True*)

Bases: `object`

Implements a custom/pure-TF Gaussian mixture model (GMM) – or mixture of Gaussians, MoG. Adaptation of parameters is conducted via the Expectation-Maximization (EM) learning algorithm and leverages full covariance matrices in the component multivariate Gaussians.

Note this is a TF wrapper model that houses the sklearn implementation for learning. The sampling process has been rewritten to utilize GPU matrix computation.

Parameters

- **`k`** – the number of components/latent variables within this GMM
- **`max_iter`** – the maximum number of EM iterations to fit parameters to data (Default = 5)
- **`assume_diag_cov`** – if True, assumes a diagonal covariance for each component (Default = False)
- **`init_kmeans`** – if True, first learn use the K-Means algorithm to initialize the component Gaussians of this GMM (Default = True)

`calc_gaussian_logpdf`(*X*)

Calculates log densities/probabilities of data X under each component given this GMM

Parameters **`X`** – the dataset to calculate the log likelihoods from

`calc_prob`(*X*)

Computes probabilities p(z|x) of data samples in X under this GMM

Parameters **`X`** – the dataset to estimate the probabilities from `calc_w_log_prob`(*X*)

Calculates weighted log probabilities of data X under each component given this GMM

Parameters `X` – the dataset to calculate the weighted log probabilities from

`e_step`(*X*)

`estimate_gaussian_parameters`(*X*, *resp*)

`estimate_log_prob`(*X*)

`fit`(*data*)

`init_from_ScikitLearn`(*gmm*)

Creates a GMM from a pre-trained Scikit-Learn model – conversion sets things up for a row-major form of sampling, i.e., s ~ mu_k + eps * (L_k^T) where k is the sampled component index

Parameters `gmm` – the pre-trained GMM (from scikit-learn) to load in

`m_step`(*X*, *log_resp*)

`predict`(*X*)

Chooses which component samples in X are likely to belong to given p(z|x)

Parameters `X` – the input data to compute p(z|x) from

`sample`(*n_s*, *mode_i=- 1*, *samples_modes_evenly=False*)

(Efficiently) Draw samples from the current underlying GMM model

Parameters

- `n_s` – the number of samples to draw from this GMM
- `mode_i` – if >= 0, will only draw samples from a specific component of this GMM (Default = -1), ignoring the Categorical prior over latent variables/components
- `samples_modes_evenly` – if True, will ignore the Categorical prior over latent variables/components and draw an approximately equal number of samples from each component

`update`(*X*)

Performs a single iterative update of parameters (assuming model initialized)

Parameters `X` – the dataset to fit this GMM to

Module contents

ngclearn.engine package

Subpackages

ngclearn.engine.cables package

Subpackages

ngclearn.engine.cables.rules package

Submodules

ngclearn.engine.cables.rules.chebb_rule module

`class ngclearn.engine.cables.rules.chebb_rule.`**`CHebbRule`**`(`*name=None*`)`

Bases: *`ngclearn.engine.cables.rules.rule.UpdateRule`*

The contrastive, bounded Hebbian update rule. Note that this rule, when used in tandem with spiking nodes and variable traces, also implements the online spike-timing dependent plasticity (STDP) rule.

Parameters **`name`** – the string name of this update rule (Default = None which creates an auto-name)

`calc_update``(`*for_bias=False*`)`

Calculates the adjustment matrix given this rule's configured internal terms

Parameters **`for_bias`** – calculate the adjustment vector (instead of a matrix) for a bias

Returns an adjustment matrix/vector

`clone``()`

`set_terms``(`*terms*, *weights=None*`)`

Sets the terms that drive this update rule

Parameters **`terms`** – list of 2-tuples where each 2-tuple is of the form (Node, string_compartment_name)

ngclearn.engine.cables.rules.hebb_rule module

`class ngclearn.engine.cables.rules.hebb_rule.`**`HebbRule`**`(`*name=None*`)`

Bases: *`ngclearn.engine.cables.rules.rule.UpdateRule`*

The Hebbian update rule.

Parameters **`name`** – the string name of this update rule (Default = None which creates an auto-name)

`calc_update``(`*for_bias=False*`)`

Calculates the adjustment matrix given this rule's configured internal terms

Parameters **`for_bias`** – calculate the adjustment vector (instead of a matrix) for a bias

Returns an adjustment matrix/vector

`clone``()`

`set_terms``(`*terms*, *weights=None*`)`

Sets the terms that drive this update rule

Parameters **`terms`** – list of 2-tuples where each 2-tuple is of the form (Node, string_compartment_name)

ngclearn.engine.cables.rules.rule module

class `ngclearn.engine.cables.rules.rule.UpdateRule`(*rule_type*, *name=None*)

Bases: `object`

Base update rule (class from which other rule types inherit basic properties from)

Parameters

- **rule_type** – the string concretely denoting this rule's type
- **name** – the string name of this update rule (Default = None which creates an auto-name)

`calc_update`(*for_bias=False*)

Calculates the adjustment matrix given this rule's configured internal terms

Parameters **for_bias** – calculate the adjustment vector (instead of a matrix) for a bias

Returns an adjustment matrix/vector

`clone`()

`point_to_cable`(*cable*, *param_name*)

Gives this update rule direct access to the source cable it will update (useful for extra statistics often required by certain local synaptic adjustment rules).

Parameters

- **cable** – the cable to point to
- **param_name** – synaptic parameters w/in this cable to point to

`set_terms`(*terms*, *weights=None*)

Sets the terms that drive this update rule

Parameters **terms** – list of 2-tuples where each 2-tuple is of the form (Node, string_compartment_name)

Module contents

Submodules

ngclearn.engine.cables.cable module

class `ngclearn.engine.cables.cable.Cable`(*cable_type*, *inp*, *out*, *name=None*, *seed=69*)

Bases: `object`

Base cable element (class from which other cable types inherit basic properties from)

Parameters

- **cable_type** – the string concretely denoting this cable's type
- **inp** – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- `out` – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

  input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- `name` – the string name of this cable (Default = None which creates an auto-name)
- `seed` – integer seed to control determinism of any underlying synapses associated with this cable

`apply_constraints()`

Apply any constraints to the learnable parameters contained within this cable.

`calc_update()`

Calculates the updates to the internal synapses that compose this cable given this cable's pre-configured synaptic update rule.

Parameters `clip_kernel` – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

`compile()`

Executes the "compile()" routine for this cable. Sub-class cables can extend this in case they contain other elements that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`get_params(`*only_learnable=False*`)`

Extract all matrix/vector parameters internal to this cable.

Parameters `only_learnable` – if True, only extracts the learnable matrix/vector parameters internal to this cable

Returns a list of matrix/vector parameters associated with this particular cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

`set_constraint(`*constraint_kernel*`)`

`set_decay(`*decay_kernel*`)`

`set_update_rule(`*preact=None*, *postact=None*, *update_rule=None*, *gamma=1.0*, *use_mod_factor=False*, *param=None*, *decay_kernel=None*`)`

Sets the synaptic adjustment rule for this cable (currently a 2-factor local synaptic Hebbian update rule).

Parameters

- `preact` – 2-Tuple defining the pre-activity/source node of which the first factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

  preact_node (Tuple[0]) the physical node that offers a pre-activity signal for the first factor of the synaptic/cable update

  preact_compartment (Tuple[1]) the component in the preact_node to extract the necessary signal to compute the first factor the synaptic/cable update

- `postact` – 2-Tuple defining the post-activity/source node of which the second factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

  postact_node (Tuple[0]) the physical node that offers a post-activity signal for the second factor of the synaptic/cable update

  postact_compartment (Tuple[1]) the component in the postact_node to extract the necessary signal to compute the second factor the synaptic/cable update

- `update_rule` – a specific update rule to use with the parameters of this cable
- `gamma` – scaling factor for the synaptic update
- `use_mod_factor` – if True, triggers the modulatory matrix weighting factor to be applied to the resultant synaptic update

  Note This is un-tested/not fully integrated

- `param` – a list of strings, each containing named parameters that are to be learned w/in this cable
- `decay_kernel` – 2-Tuple defining the type of weight decay to be applied to the synapses. The value types inside each slot of the tuple are specified below:

  decay_type (Tuple[0]) string indicating which type of weight decay to use, "l2" will trigger L2-penalty decay, while "l1" will trigger L1-penalty decay

  decay_coefficient (Tuple[1]) scalar/float to control magnitude of decay applied to computed local updates

ngclearn.engine.cables.dcable module

`class ngclearn.engine.cables.dcable.`**`DCable`**(*inp*, *out*, *init_kernels=None*, *shared_param_path=None*, *clip_kernel=None*, *constraint_kernel=None*, *seed=69*, *name=None*)

Bases: `ngclearn.engine.cables.cable.Cable`

A dense cable that transforms signals that travel across via a bundle of synapses. (In other words, a linear projection followed by an optional base-rate/bias shift.)

Note: a dense cable only contains two possible learnable parameters, "A" and "b" each with only two terms for their local Hebbian updates.

Parameters

- `inp` – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- `out` – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- `w_kernel` – an N-Tuple defining type of scheme to randomly initialize weights.

  scheme (Tuple[0]) triggers the type of initalization scheme, for example, "gaussian" will apply an elementwise Gaussian initialization. (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their string codes that can be used.)

  scheme_arg1 (Tuple[1]) first argument to control the initialization (for many schemes, setting this value to 1.0 or even omitting it is acceptable given that this parameter is ignored, for example, in "unif_scale", the second argument would be ignored.) (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their extra arguments.)

  scheme_arg2 (Tuple[2]) second argument to control the initialization – this is generally only necessary to set in the case of lateral competition initialization schemes, such as in the case of "lkwta" which requires a 3-Tuple specified as follows: ("lkwta",alpha_scale,beta_scale) where alpha_scale controls the strength of self-excitation and beta_scale controls the strength of the cross-unit inhibition.

- `b_kernel` – 2-Tuple defining type of scheme to randomly initialize weights.

  scheme (Tuple[0]) triggers the type of initalization scheme, for example, "gaussian" will apply an elementwise Gaussian initialization. (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their string codes that can be used.)

  scheme_arg1 (Tuple[1]) first argument to control the initialization (for many schemes, setting this value to 1.0 or even omitting it is acceptable given that this parameter is ignored, for example, in "unif_scale", the second argument would be ignored.) (See the documentation for init_weights() in ngclearn.utils.transform_utils for details on all the types of initializations and their extra arguments.)

- `shared_param_path` –
- `clip_kernel` – 3-Tuple defining type of clipping to apply to calculated synaptic adjustments.

  clip_type (Tuple[0]) type of clipping constraint to apply. If "hard_clip" is set, then a hard-clipping routine is applied (ignoring "clip_axis") while "norm_clip" clips by checking if the norm exceeds "clip_value" along "clip_axis". Note that "hard_clip" will also be applied to biases (while "clip_norm" is not).

  clip_value (Tuple[1]) the magnitude of the worse-case bounds of the clip to apply/enforce.

  clip_axis (Tuple[2]) the axis along which the clipping is to be applied (to each matrix).

  Note specifying None will mean no clipping is applied to this cable's calculated updates

- `constraint_kernel` – Dict defining the constraint type to be applied to the learnable parameters of this cable. The expected keys and corresponding value types are specified below:

'clip_type' type of clipping constraint to be applied to learnable parameters/synapses. If "norm_clip" is specified, then norm-clipping will be applied (with a check if the norm exceeds "clip_mag"), and if "forced_norm_clip" then norm-clipping will be applied regardless each time apply_constraint() is called.

'clip_mag' the magnitude of the worse-case bounds of the clip to apply/enforce.

'clip_axis' the axis along which the clipping is to be applied (to each matrix).

Note specifying None will mean no constraints are applied to this cable's parameters

- `name` – the string name of this cable (Default = None which creates an auto-name)
- `seed` – integer seed to control determinism of any underlying synapses associated with this cable

`apply_constraints()`

Apply any constraints to the learnable parameters contained within
this cable. This function will execute any of the following
pre-configured constraints:
1) project weights to adhere to vector norm constraints
2) apply weight decay (to non-bias synaptic matrices)

`calc_update()`

Calculates the updates to the internal synapses that compose this cable given this cable's pre-configured synaptic update rule.

Parameters `clip_kernel` – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

`compile()`

Executes the "compile()" routine for this cable.

Returns a dictionary containing post-compilation check information about this cable

`get_params`(*only_learnable=False*)

Extract all matrix/vector parameters internal to this cable.

Parameters `only_learnable` – if True, only extracts the learnable matrix/vector parameters internal to this cable

Returns a list of matrix/vector parameters associated with this particular cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

`set_update_rule`(*preact=None*, *postact=None*, *update_rule=None*, *gamma=1.0*, *use_mod_factor=False*, *param=None*, *decay_kernel=None*)

Sets the synaptic adjustment rule for this cable (currently a 2-factor local synaptic Hebbian update rule).

Parameters

- `preact` – 2-Tuple defining the pre-activity/source node of which the first factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

preact_node (Tuple[0]) the physical node that offers a pre-activity signal for the first factor of the synaptic/cable update

preact_compartment (Tuple[1]) the component in the preact_node to extract the necessary signal to compute the first factor the synaptic/cable update

- `postact` – 2-Tuple defining the post-activity/source node of which the second factor the synaptic update rule will be extracted from. The value types inside each slot of the tuple are specified below:

  postact_node (Tuple[0]) the physical node that offers a post-activity signal for the second factor of the synaptic/cable update

  postact_compartment (Tuple[1]) the component in the postact_node to extract the necessary signal to compute the second factor the synaptic/cable update

- `update_rule` – a specific update rule to use with the parameters of this cable
- `gamma` – scaling factor for the synaptic update
- `use_mod_factor` – if True, triggers the modulatory matrix weighting factor to be applied to the resultant synaptic update

  Note This is un-tested/not fully integrated

- `param` – a list of strings, each containing named parameters that are to be learned w/in this cable
- `decay_kernel` – 2-Tuple defining the type of weight decay to be applied to the synapses. The value types inside each slot of the tuple are specified below:

  decay_type (Tuple[0]) string indicating which type of weight decay to use, "l2" will trigger L2-penalty decay, while "l1" will trigger L1-penalty decay

  decay_coefficient (Tuple[1]) scalar/float to control magnitude of decay applied to computed local updates

ngclearn.engine.cables.scable module

`class ngclearn.engine.cables.scable.SCable(`*inp*, *out*, *coeff=1.0*, *name=None*, *seed=69*`)`

Bases: `ngclearn.engine.cables.cable.Cable`

A simple cable that, at most, applies a scalar amplification of signals that travel across it. (Otherwise, this cable works like an identity carry-over.)

Parameters

- `inp` – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the source/input Node object that this cable will carry signal information from

  input_compartment (Tuple[1]) the compartment within the source/input Node that signals will extracted and transmitted from

- `out` – 2-Tuple defining the nodal points that this cable will connect. The value types inside each slot of the tuple are specified below:

  input_node (Tuple[0]) the destination/output Node object that this cable will carry signal information to

input_compartment (Tuple[1]) the compartment within the destination/output Node that signals transmitted and deposited into

- `coeff` – a scalar float to control any signal scaling associated with this cable
- `name` – the string name of this cable (Default = None which creates an auto-name)
- `seed` – integer seed to control determinism of any underlying synapses associated with this cable

`compile()`

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`propagate()`

Internal transmission function that computes the correct transformation of a source node to a destination node

Returns the resultant transformed signal (transformation f information from "node")

Module contents

ngclearn.engine.nodes package

Submodules

ngclearn.engine.nodes.enode module

class ngclearn.engine.nodes.enode.**ENode**(*name*, *dim*, *error_type='mse'*, *act_fx='identity'*, *batch_size=1*, *precis_kernel=None*, *constraint_kernel=None*, *ex_scale=1.0*)

Bases: *ngclearn.engine.nodes.node.Node*

Implements a (rate-coded) error node simplified to its fixed-point form:

e = target - mu // in the case of squared error (Gaussian error units)

e = signum(target - mu) // in the case of absolute error (Laplace error units)

where:

target - a desired target activity value (target = pred_targ)

mu - an external prediction signal of the target activity value (mu = pred_mu)

Compartments:

* pred_mu - prediction signals (deposited signals summed)
* pred_targ - target signals (deposited signals summed)
* z - the error neural activities, set as z = e
* phi(z) - the post-activation of the error activities in z
* L - the local loss represented by the error activities
* avg_scalar - multiplies L and z by (1/avg_scalar)

Parameters

- `name` – the name/label of this node
- `dim` – number of neurons this node will contain/model
- `error_type` – type of distance/error measured by this error node. Setting this to "mse" will set up squared-error neuronal units (derived from L = 0.5 * (Sum_j (target - mu)^2_j)), and "mae" will set up mean absolute error neuronal units (derived from L = Sum_j |target - mu|).
- `act_fx` – activation function – phi(v) – to apply to error activities (Default = "identity")
- `batch_size` – batch-size this node should assume (for use with static graph optimization)
- `precis_kernel` – 2-Tuple defining the initialization of the precision weighting synapses that will modulate the error neural activities. For example, an argument could be: ("uniform", 0.01) The value types inside each slot of the tuple are specified below:

  init_scheme (Tuple[0]) initialization scheme, e.g., "uniform", "gaussian".

  init_scale (Tuple[1]) scalar factor controlling the scale/magnitude of initialization distribution, e.g., 0.01.

  Note specifying None will result in precision weighting being applied to the error neurons. Understand that care should be taken w/ respect to this particular argument as precision synapses involve an approximate inversion throughout simulation steps

- `constraint_kernel` – Dict defining the constraint type to be applied to the learnable parameters of this node. The expected keys and corresponding value types are specified below:

  'clip_type' type of clipping constraint to be applied to learnable parameters/synapses. If "norm_clip" is specified, then norm-clipping will be applied (with a check if the norm exceeds "clip_mag"), and if "forced_norm_clip" then norm-clipping will be applied regardless each time apply_constraint() is called.

  'clip_mag' the magnitude of the worse-case bounds of the clip to apply/enforce.

  'clip_axis' the axis along which the clipping is to be applied (to each matrix).

  Note specifying None will mean no constraints are applied to this node's parameters

- `ex_scale` – a scale factor to amplify error neuron signals (Default = 1)

`apply_constraints()`

Apply any constraints to the learnable parameters contained within
this cable. This function will execute any of the following
pre-configured constraints:
1) compute new precision matrices
2) project synapses to adhere to any embedded norm constraints

`calc_update(`*update_radius=- 1.0*`)`

Calculates the updates to local internal synaptic parameters related to this specific node given current relevant values (such as node-level precision matrices).

Parameters `update_radius` – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm) `compile()`

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`compute_precision`(*rebuild_cov=True*)

Co-function that pre-computes the precision matrices for this NGC node. NGC uses the Cholesky-decomposition form of precision (Sigma)^{-1}

Parameters **`rebuild_cov`** – rebuild the underlying covariance matrix after re-computing precision (Default = True)

`set_constraint`(*constraint_kernel*)

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **`injection_table`** –
- **`skip_core_calc`** – skips the core components of this node's calculation (Default = False)

ngclearn.engine.nodes.fnode module

class ngclearn.engine.nodes.fnode.**FNode**(*name*, *dim*, *act_fx='identity'*, *batch_size=1*)

Bases: *ngclearn.engine.nodes.node.Node*

Implements a feedforward (stateless) transmission node:

z = dz

where:

dz - aggregated input signals from other nodes/locations

Compartments:

* dz - incoming pressures/signals (deposited signals summed)
* z - the state values/neural activities, set as: z = dz
* phi(z) - the post-activation of the neural activities

Parameters

- **`name`** – the name/label of this node
- **`dim`** – number of neurons this node will contain/model
- **`act_fx`** – activation function – phi(v) – to apply to neural activities
- **`batch_size`** – batch-size this node should assume (for use with static graph optimization)

`compile()`

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **`injection_table`** –
- **`skip_core_calc`** – skips the core components of this node's calculation (Default = False)

ngclearn.engine.nodes.node module

class ngclearn.engine.nodes.node.**Node**(*node_type*, *name*, *dim*)

Bases: `object`

Base node element (class from which other node types inherit basic properties from)

Parameters

- **`node_type`** – the string concretely denoting this node's type
- **`name`** – str name of this node
- **`dim`** – number of neurons this node will contain

`calc_update`(*update_radius=- 1.0*)

Calculates the updates to local internal synaptic parameters related to this specific node given current relevant values (such as node-level precision matrices).

Parameters **`update_radius`** – radius of Gaussian ball to constrain computed update matrices by (i.e., clipping by Frobenius norm)

`clamp`(*data*, *is_persistent=True*)

Clamps an externally provided named value (a vector/matrix) to the desired compartment within this node.

Parameters

- **`data`** – 2-Tuple containing a named external signal to clamp

  compartment_name (Tuple[0]) the (str) name of the compartment to clamp this data signal to.

  signal (Tuple[1]) the data signal block to clamp to the desired compartment name

- **`is_persistent`** – if True, prevents this node from overriding the clamped data over time (Default = True)

`clear`()

Wipes/clears values of each compartment in this node (and sets .is_clamped = False).

`compile`()

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable `deep_store_state()`

Performs a deep copy of all compartment statistics.

Returns Dict containing a deep copy of each named compartment of this node

`extract(`*comp_name*`)`

Extracts the data signal value that is currently stored inside of a target compartment

Parameters `comp_name` – the name of the compartment in this node to extract data from

`extract_params()`

`inject(`*data*`)`

Injects an externally provided named value (a vector/matrix) to the desired compartment within this node.

Parameters `data` – 2-Tuple containing a named external signal to clamp

compartment_name (Tuple[0]) the (str) name of the compartment to clamp this data signal to.

signal (Tuple[1]) the data signal block to clamp to the desired compartment name

`set_cold_state(`*injection_table=None*, *batch_size=- 1*`)`

Sets each compartment to its cold zero-state of shape (batch_size x D). Note that this fills each vector/matrix state of each compartment to all zero values.

Parameters

- `injection_table` –
- `batch_size` – the axis=0 dimension of each compartment @ its cold zero-state

`set_constraint(`*constraint_kernel*`)`

`set_status(`*status=('static', 1)*`)`

Sets the status of this node to be either "static" or "dynamic".

Note: Making this node "dynamic" in the sense that it can handle mini-batches of samples of arbitrary length BUT you CANNOT use "use_graph_optim = True" static-graph acceleration used w/in the NGC-Graph .settle() routine, meaning your simulation run slower than when using acceleration.

Parameters `status` – 2-tuple where 1st element contains a string status flag and the 2nd element contains the (integer) batch_size. If status is set to "dynamic", the second argument is arbitrary (setting it to 1 is sufficient), and if status is set to "static" you MUST choose a fixed batch_size that you will use w.r.t. this node.

`step(`*injection_table=None*, *skip_core_calc=False*`)`

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- `injection_table` –
- `skip_core_calc` – skips the core components of this node's calculation (Default = False)

`wire_to(`*dest_node*, *src_comp*, *dest_comp*, *cable_kernel=None*, *mirror_path_kernel=None*, *name=None*, *short_name=None*`)`

A wiring function that connects this node to another external node via a cable (or synaptic bundle)

Parameters

- `dest_node` – destination node (a Node object) to wire this node to
- `src_comp` – name of the compartment inside this node to transmit a signal from (to destination node)
- `dest_comp` – name of the compartment inside the destination node to transmit a signal to
- `cable_kernel` – Dict defining how to initialize the cable that will connect this node to the destination node. The expected keys and corresponding value types are specified below:

  'type' type of cable to be created. If "dense" is specified, a DCable (dense cable/bundle/matrix of synapses) will be used to transmit/transform information along.

  'init_kernels' a Dict specifying how parameters w/in the learnable parts of the cable are to randomly initialized

  'seed' integer seed to deterministically control initialization of synapses in a DCable

  Note either cable_kernel, mirror_path_kernel MUST be set to something that is not None

- `mirror_path_kernel` – 2-Tuple that allows a currently existing cable to be re-used as a transformation. The value types inside each slot of the tuple are specified below:

  cable_to_reuse (Tuple[0]) target cable (usually an existing DCable object) to shallow copy and mirror

  mirror_type (Tuple[1]) how should the cable be mirrored? If "symm_tied" is specified, then the transpose of this cable will be used to transmit information from this node to a destination node, if "anti_symm_tied" is specified, the negative transpose of this cable will be used, and if "tied" is specified, then this cable will be used exactly in the same way it was used in its source cable.

  Note either cable_kernel, mirror_path_kernel MUST be set to something that is not None

- `name` – the string name to be assigned to the generated cable (Default = None)

  Note setting this to None will trigger the created cable to auto-name itself

ngclearn.engine.nodes.snode module

class ngclearn.engine.nodes.snode.**SNode**(*name*, *dim*, *beta=1.0*, *leak=0.0*, *zeta=1.0*, *act_fx='identity'*, *batch_size=1*, *integrate_kernel=None*, *prior_kernel=None*, *threshold_kernel=None*, *trace_kernel=None*, *samp_fx='identity'*)

Bases: `ngclearn.engine.nodes.node.Node`

Implements a (rate-coded) state node that follows NGC settling dynamics according to:

d.z/d.t = -z * leak + dz + prior(z), where dz = dz_td + dz_bu * phi'(z)

where:

dz - aggregated input signals from other nodes/locations

leak - controls strength of leak variable/decay prior(z) - distributional prior placed over z (such as a kurtotic prior)

Note that the above is used to adjust neural activity values via an integator inside a node. For example, if the standard/default Euler integrator is used then the neurons inside this node are adjusted per step as follows:

z <- z * zeta + d.z/d.t * beta

where:

beta - strength of update to node state z

zeta - controls the strength of recurrent carry-over, if set to 0 no carry-over is used (stateless)

Compartments:

* dz_td - the top-down pressure compartment (deposited signals summed)

* dz_bu - the bottom-up pressure compartment, potentially weighted by phi'(x)) (deposited signals summed)

* z - the state neural activities

* phi(z) - the post-activation of the state activities

* S(z) - the sampled state of phi(z) (Default = identity or f(phi(z)) = phi(z))

* mask - a binary mask to be applied to the neural activities

Parameters

- **name** – the name/label of this node
- **dim** – number of neurons this node will contain/model
- **beta** – strength of update to adjust neurons at each simulation step (Default = 1)
- **leak** – strength of the leak applied to each neuron (Default = 0)
- **zeta** – effect of recurrent/stateful carry-over (Defaul = 1)
- **act_fx** – activation function – phi(v) – to apply to neural activities

  Note if using either "kwta" or "bkwta", please input how many winners should win the competiton, i.e., use "kwta(N)" or "bkwta(N)" where N is an integer > 0.
- **batch_size** – batch-size this node should assume (for use with static graph optimization)
- **integrate_kernel** – Dict defining the neural state integration process type. The expected keys and corresponding value types are specified below:

  'integrate_type' type integration method to apply to neural activity over time. If "euler" is specified, Euler integration will be used (future ngc-learn versions will support "midpoint"/other methods).

  'use_dfx' a boolean that decides if phi'(v) (activation derivative) is used in the integration process/update.

  Note specifying None will automatically set this node to use Euler integration w/ use_dfx=False
- **prior_kernel** – Dict defining the type of prior function to apply over neural activities. The expected keys and corresponding value types are specified below:

  'prior_type' type of (centered) distribution to use as a prior over neural activities. If "laplace" is specified, a Laplacian distribution is used, if "cauchy" is specified, a Cauchy distribution will be used, if "gaussian" is specified, a Gaussian distribution will be used, and if "exp" is specified, the exponential distribution will be used.

'lambda' the scale factor controlling the strength of the prior applied to neural activities.

Note specifying None will result in no prior distribution being applied

- `threshold_kernel` – Dict defining the type of threshold function to apply over neural activities. The expected keys and corresponding value types are specified below:

  'threshold_type' type of (centered) distribution to use as a prior over neural activities. If "soft_threshold" is specified, a soft thresholding function is used, and if "cauchy_threshold" is specified, a cauchy thresholding function is used,

  'thr_lambda' the scale factor controlling the strength of the threshold applied to neural activities.

  Note specifying None will result in no threshold function being applied

- `trace_kernel` – <unused> (Default = None)
- `samp_fx` – the sampling/stochastic activation function – S(v) – to apply to neural activities (Default = identity)

`compile()`

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- `injection_table` –
- `skip_core_calc` – skips the core components of this node's calculation (Default = False)

ngclearn.engine.nodes.spnode_lif module

`class ngclearn.engine.nodes.spnode_lif.SpNode_LIF`(*name*, *dim*, *batch_size=1*, *integrate_kernel=None*, *spike_kernel=None*, *trace_kernel=None*)

Bases: *ngclearn.engine.nodes.node.Node*

Implements a leaky-integrate and fire (LIF) spiking state node that follows NGC settling dynamics according to:

Jz = dz OR d.Jz/d.t = (-Jz + dz) * (dt/tau_curr) IF zeta > 0 // current

d.Vz/d.t = (-Vz + Jz * R) * (dt/tau_mem) // voltage

spike(t) = spike_response_model(Jz(t), Vz(t), ref(t)…) // spikes computed according to SRM

trace(t) = (trace(t-1) * alpha) * (1 - Sz(t)) + Sz(t) // variable trace filter

where:

Jz - current value of the electrical current input to the spiking neurons w/in this node

Vz - current value of the membrane potential of the spiking neurons w/in this node
Sz - the spike signal reading(s) of the spiking neurons w/in this node
dz - aggregated input signals from other nodes/locations to drive current Jz
ref - the current value of the refractory variables (accumulates with time and forces neurons to rest)
alpha - variable trace's interpolation constant (dt/tau <– input by user)
tau_mem - membrane potential time constant (R_m * C_m - resistance times capacitance)
tau_curr - electrical current time constant strength
dt - the integration time constant d.t
R - neural membrane resistance

Note that the above is used to adjust neural electrical current values via an integator inside a node. For example, if the standard/default Euler integrator is used then the neurons inside this node are adjusted per step as follows:
Jz <- Jz + d.Jz/d.t // <– only if zeta > 0
Vz <- Vz + d.Vz/d.t
ref <- ref + d.t (resets to 0 after 1 millisecond)

Compartments:
* dz_td - the top-down pressure compartment (deposited signals summed)
* dz_bu - the bottom-up pressure compartment, potentially weighted by phi'(x)) (deposited signals summed)
* Jz - the neural electrical current values
* Vz - the neural membrane potential values
* Sz - the current spike values (binary vector signal) at time t
* Trace_z - filtered trace values of the spike values (real-valued vector)
* ref - the refractory variables (an accumulator)
* mask - a binary mask to be applied to the neural activities

Constants:
* V_thr -
* dt - the integration time constant (milliseconds)
* R - the neural membrane resistance (mega Ohms)
* C - the neural membrane capacitance (microfarads)
* tau_m - the membrane potential time constant (tau_m = R * C)
* tau_c - the electrial current time constant (if zeta > 0)
* trace_alpha - the trace variable's interpolation constant
* ref_T - the length of the absolute refractory period (milliseconds)

Parameters

- `name` – the name/label of this node
- `dim` – number of neurons this node will contain/model
- `batch_size` – batch-size this node should assume (for use with static graph optimization)
- `integrate_kernel` – Dict defining the neural state integration process type. The expected keys and corresponding value types are specified below:

'integrate_type' type integration method to apply to neural activity over time. If "euler" is specified, Euler integration will be used (future ngc-learn versions will support "midpoint"/other methods).

'use_dfx' <UNUSED>

'dt' type integration time constant for the spiking neurons

Note specifying None will automatically set this node to use Euler integration with dt = 0.25 ms

- **spike_kernel** – Dict defining the properties of the spiking process. The expected keys and corresponding value types are specified below:

  'V_thr' the (constant) voltage threshold a neuron must cross to spike

  'zeta' a trigger variable - if > 0, electrical current will be integrated over as well

  'tau_mem' the membrane potential time constant

  'tau_curr' the electrical current time constant (only used if zeta > 0, otherwise ignored)

- **trace_kernel** – Dict defining the signal tracing process type. The expected keys and corresponding value types are specified below:

  'dt' type integration time constant for the trace

  'tau' the filter time constant for the trace

  Note specifying None will automatically set this node to not use variable tracing

compile()

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

step(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **injection_table** –
- **skip_core_calc** – skips the core components of this node's calculation (Default = False)

ngclearn.engine.nodes.spnode_enc module

class ngclearn.engine.nodes.spnode_enc.**SpNode_Enc**(*name*, *dim*, *gain=1.0*, *batch_size=1*, *trace_kernel=None*)

Bases: `ngclearn.engine.nodes.node.Node`

Implements a simple spiking state node that converts its real-valued input
vector into an on-the-fly generated Poisson spike train. To control the
firing frequency of the spiking neurons within this model, modify the

gain parameter (range [0,1]) – for example, on pixel data normalized
to the range of [0,1], setting the gain to 0.25 will result in a firing
frequency of approximately 63.75 Hertz (Hz). Note that for real-valued data
which should be normalized to the range of [0,1], the actual values of each
dimension will be used to dictate specific spiking rates (each dimension
spikes in proportion to its feature value/probability).

Compartments:
* z - the real-valued input variable to convert to spikes (should be clamped)
* Sz - the current spike values (binary vector signal) at time t
* Trace_z - filtered trace values of the spike values (real-valued vector)
* mask - a binary mask to be applied to the neural activities

Parameters

- `name` – the name/label of this node
- `dim` – number of neurons this node will contain/model
- `leak` – strength of the conductance leak applied to each neuron's current Jz (Default = 0)
- `batch_size` – batch-size this node should assume (for use with static graph optimization)
- `trace_kernel` – Dict defining the signal tracing process type. The expected keys and corresponding value types are specified below:

  'dt' type integration time constant for the trace

  'tau' the filter time constant for the trace

  Note specifying None will automatically set this node to not use variable tracing

`compile()`

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

`step`(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- `injection_table` –
- `skip_core_calc` – skips the core components of this node's calculation (Default = False)

ngclearn.engine.nodes.fnode_ba module

class ngclearn.engine.nodes.fnode_ba.**FNode_BA**(*name*, *dim*, *act_fx='identity'*, *batch_size=1*)

Bases: *ngclearn.engine.nodes.node.Node*

This is a "teaching" forward node - this node was particularly designed to
generate a learning signal for models that are to learn via a broadcast
(feedback) alignment approach. (This is a convenience class to support BA.)

Implements a feedforward teaching (stateless) transmission node:
    z = (dz) * [(Jz > 0) * sech^2(Jz * c2)] and z = dz if c2 <= 0
where:
    dz - aggregated input signals from other nodes/locations (typically error nodes)
    Jz - input current signal
    c2 - a constant value tuned to the sech^2(x) function (this is for a spiking neurons)
        note that setting c2 to <= 0, then a non-spiking teaching forward node signal
        is created (note if c2 = 0, then this node does not use its Jz compartment)

Compartments:
    * dz - incoming pressures/signals (deposited signals summed)
    * Jz - input current signal to use to trigger spiking BA-learning
    * z - the state values/neural activities, set as: z = dz
    * phi(z) - the post-activation of the neural activities

Parameters

- **name** – the name/label of this node
- **dim** – number of neurons this node will contain/model
- **act_fx** – activation function – phi(v) – to apply to neural activities
- **batch_size** – batch-size this node should assume (for use with static graph optimization)

compile()

Executes the "compile()" routine for this node. Sub-class nodes can extend this in case they contain other elements besides compartments that must be configured properly for global simulation usage.

Returns a dictionary containing post-compilation check information about this cable

step(*injection_table=None*, *skip_core_calc=False*)

Executes this nodes internal integration/calculation for one discrete step in time, i.e., runs simulation of this node for one time step.

Parameters

- **injection_table** –
- **skip_core_calc** – skips the core components of this node's calculation (Default = False)

Module contents

Submodules

ngclearn.engine.ngc_graph module

`class ngclearn.engine.ngc_graph.`**`NGCGraph`**`(`*K=5*, *name='ncn'*, *batch_size=1*`)`

Bases: `object`

Implements the full model structure/graph for an NGC system composed of nodes and cables. Note that when instantiating this object, it is important to call .compile(), like so:

graph = NGCGraph(. . .)
info = graph.compile()

Parameters

- **`K`** – number of iterative inference/settling steps to simulate
- **`name`** – (optional) the name of this projection graph (Default="ncn")
- **`batch_size`** – fixed batch-size that the underlying compiled static graph system should assume (Note that you can set this also as an argument to .compile())

  Note if "use_graph_optim" is set to False, then this argument is not meaningful as the system will work with variable-length batches

`apply_constraints``()`

Apply any constraints to the signals embedded in this graph. This function will execute any of the following pre-configured constraints:

1) compute new precision matrices (if applicable)

2) project weights to adhere to vector norm constraints

`calc_updates``(`*debug_map=None*`)`

Calculates the updates to synaptic weight matrices along each learnable wire within this graph via a generalized Hebbian learning rule.

Parameters **`debug_map`** – (Default = None), a Dict to place named signals inside (for debugging)

`clamp``(`*clamp_targets*`)`

Clamps an externally provided named value (a vector/matrix) to the desired compartment within a particular Node of this NGC graph. Note that clamping means this value typically means the value clamped on will persist (it will NOT evolve according to the injected node's dynamics over simulation steps, unless is_persistent = True).

Parameters **`clamp_targets`** – 3-Tuple containing a named external signal to clamp

node_name (Tuple[0]) the (str) name of the node to clamp a data signal to.

compartment_name (Tuple[1]) the (str) name of the node's compartment to clamp this data signal to.

signal (Tuple[2]) the data signal block to clamp to the desired compartment name `clear()`

Clears/deletes any persistent signals currently embedded w/in this graph's Nodes

`clone_state()`

Clones the entire state of this graph (in terms of signals/tensors) and stores each node's state dictionary in global has map

Returns a Dict (hash table) containing string names that map to physical Node objects

`compile`(*use_graph_optim=True*, *batch_size=- 1*)

Executes a global "compile" of this simulation object to ensure internal system coherence. (Only call this function after the constructor has been set).

Parameters

- **use_graph_optim** – if True, this simulation will use static graph acceleration (Default = True)
- **batch_size** – if > 0, will set the integer global batch_size of this simulation object (otherwise, self.batch_size will be used)

Returns a dictionary containing post-compilation information about this simulation object

`evolve`(*clamped_vars=None*, *readout_vars=None*, *init_vars=None*, *cold_start=True*, *K=- 1*, *masked_vars=None*)

Evolves this simulation object for one full K-step episode given input information through clamped and initialized variables. Note that this is a *convenience function* written to embody an NGC system's full settling process, its local synaptic update calculations, as well as the optimization of and application of constraints to the synaptic parameters contained within .theta.

Parameters

- **clamped_vars** – list of 3-tuple strings containing named Nodes, their compartments, and values to (persistently) clamp on. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),…]
- **readout_vars** – list of 2-tuple strings containing Nodes and their compartments to read from (in this function's output). Note that this list takes the form: [(node1_name, node1_compartment), node2_name, node2_compartment),…]
- **init_vars** – list of 3-tuple strings containing named Nodes, their compartments, and values to initialize each Node from. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),…]
- **cold_start** – initialize all non-clamped/initialized Nodes (i.e., their compartments contain None) to zero-vector starting points
- **K** – number simulation steps to run (Default = -1), if <= 0, then self.K will be used instead
- **masked_vars** – list of 4-tuple that instruct which nodes/compartments/masks/clamped values to apply. This list is used to trigger auto-associative recalls from this NGC graph. Note that this list takes the form: [(node1_name, node1_compartment, mask, value), node2_name, node2_compartment, mask, value),…]

Returns

readouts, delta; where "readouts" is a 3-tuple list of the form [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),…]

`extract`(*node_name*, *node_var_name*)

Extract a particular signal from a particular node embedded in this graph

Parameters

- `node_name` – name of the node from the NGC graph to examine
- `node_var_name` – compartment name w/in Node to extract signal from

Returns an extracted signal (vector/matrix) OR None if either the node does not exist or the entire system has not been simulated (meaning that no node dynamics have been run yet)

`getNode`(*node_name*)

Extract a particular node from this graph

Parameters `node_name` – name of the node from the NGC graph to examine

Returns the desired Node (object) or None if the node does not exist

`inject`(*injection_targets*)

Injects an externally provided named value (a vector/matrix) to the desired compartment within a particular Node of this NGC graph. Note that injection means this value does not persist (it will evolve according to the injected node's dynamics over simulation steps).

Parameters

- `injection_targets` – 3-Tuple containing a named external signal to clamp

  node_name (Tuple[0]) the (str) name of the node to clamp a data signal to.

  compartment_name (Tuple[1]) the (str) name of the compartment to clamp this data signal to.

  signal (Tuple[2]) the data signal block to clamp to the desired compartment name

- `is_persistent` – if True, clamped data value will persist throughout simulation (Default = True)

`parse_node_values`(*node_values*)

`set_cycle`(*nodes*, *param_order=None*)

Set an execution cycle in this graph

Parameters `nodes` – an ordered list of Node(s) to create an execution cycle for

`set_learning_order`(*param_order*)

Forces this simulation object to arrange its .theta and delta to follow a particular order.

Parameters `param_order` – a list of Cables/Nodes which will dictate the strict order in which parameter updates will be calculated and how they are arranged in .theta (note that delta and theta will match the same dictated order)

`set_optimization`(*opt_algo*)

Sets the internal optimization algorithm used by this simulation object.

Parameters `opt_algo` – optimization algorithm to be used, e.g., SGD, Adam, etc. (Note: must be a valid TF2 optimizer.)

`set_theta`(*theta_target*)

`set_to_resting_state`(*batch_size=- 1*)

`set_to_state`(*state_map*)

Set every state of every node in this graph to the values contained in the global Dict (hash table) “state_map”

Parameters `state_map` – a Dict (hash table) containing string names that map to physical Node objects

`settle`(*clamped_vars=None*, *readout_vars=None*, *init_vars=None*, *cold_start=True*, *K=- 1*, *debug=False*, *masked_vars=None*, *calc_delta=True*)

Execute this NGC graph’s iterative inference using the execution pathway(s) defined at construction/initialization.

Parameters

- `clamped_vars` – list of 3-tuple strings containing named Nodes, their compartments, and values to (persistently) clamp on. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]
- `readout_vars` – list of 2-tuple strings containing Nodes and their compartments to read from (in this function’s output). Note that this list takes the form: [(node1_name, node1_compartment), node2_name, node2_compartment),. . .]
- `init_vars` – list of 3-tuple strings containing named Nodes, their compartments, and values to initialize each Node from. Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]
- `cold_start` – initialize all non-clamped/initialized Nodes (i.e., their compartments contain None) to zero-vector starting points/resting states
- `K` – number simulation steps to run (Default = -1), if <= 0, then self.K will be used instead
- `debug` – <UNUSED>
- `masked_vars` – list of 4-tuple that instruct which nodes/compartments/masks/clamped values to apply. This list is used to trigger auto-associative recalls from this NGC graph. Note that this list takes the form: [(node1_name, node1_compartment, mask, value), node2_name, node2_compartment, mask, value),. . .]
- `calc_delta` – compute the list of synaptic updates for each learnable parameter within .theta? (Default = True)

Returns

readouts, delta; where “readouts” is a 3-tuple list of the form [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .], and “delta” is a list of synaptic adjustment matrices (in the same order as .theta)

`step`(*calc_delta=False*)

Online function for simulating exactly one discrete time step of this simulated NGC graph given its exact current state.

Parameters `calc_delta` – compute the list of synaptic updates for each learnable parameter within .theta? (Default = True)

Returns

readouts, delta; where "readouts" is a 3-tuple list of the form [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .], and "delta" is a list of synaptic adjustment matrices (in the same order as .theta)

ngclearn.engine.proj_graph module

class ngclearn.engine.proj_graph.**ProjectionGraph**(*name='sampler'*)

Bases: `object`

Implements a projection graph – useful for conducting ancestral sampling of a directed generative model or ancestral projection of a clamped graph. Note that when instantiating this object, it is important to call .compile(), like so:

graph = ProjectionGraph(. . .)

info = graph.compile()

Parameters **name** – the name of this projection graph

apply_constraints()

Apply any constraints to the signals embedded in this graph. This function will execute any of the following pre-configured constraints:

1) compute new precision matrices (if applicable)

2) project weights to adhere to vector norm constraints

calc_updates(*debug_map=None*)

Calculates the updates to synaptic weight matrices along each learnable wire within this NCN operation graph via a generalized Hebbian learning rule.

Parameters **debug_map** – (Default = None), a Dict to place named signals inside (for debugging)

clamp(*clamp_targets*)

Clamps an externally provided named value (a vector/matrix) to the desired compartment within a particular Node of this projection graph.

Parameters **clamp_targets** – 3-Tuple containing a named external signal to clamp

node_name (Tuple[0]) the (str) name of the node to clamp a data signal to.

compartment_name (Tuple[1]) the (str) name of the node's compartment to clamp this data signal to.

signal (Tuple[2]) the data signal block to clamp to the desired compartment name

clear()

Clears/deletes any persistent signals currently embedded w/in this graph's Nodes

compile(*batch_size=- 1*)

Executes a global "compile" of this simulation object to ensure internal system coherence. (Only call this function after the constructor has been set).

Parameters **batch_size** – <UNUSED>

Returns a dictionary containing post-compilation information about this simulation object `extract`(*node_name*, *node_var_name*)

Extract a particular signal from a particular node embedded in this graph

Parameters

- **node_name** – name of the node from the NGC graph to examine
- **node_var_name** – compartment name w/in Node to extract signal from

Returns an extracted signal (vector/matrix) OR None if node does not exist

`getNode`(*node_name*)

Extract a particular node from this graph

Parameters **node_name** – name of the node from the NGC graph to examine

Returns the desired Node (object)

`project`(*clamped_vars=None*, *readout_vars=None*)

Project signals through the execution pathway(s) defined by this graph

Parameters

- **clamped_vars** – list of 2-tuples containing named Nodes that will be clamped with particular values. Note that this list takes the form: [(node1_name, node_value1), node2_name, node_value2),. . .]
- **readout_vars** – list of 2-tuple strings containing named Nodes and their compartments to read signals from. Note that this list takes the form: [(node1_name, node1_compartment), node2_name, node2_compartment),. . .]

Returns

readout values - a list of 3-tuples named signals corresponding to the ones in "readout_vars". Note that this list takes the form: [(node1_name, node1_compartment, value), node2_name, node2_compartment, value),. . .]

`set_cold_state`(*batch_size=- 1*)

`set_cycle`(*nodes*)

Set execution cycle for this graph

Parameters **nodes** – an ordered list of Node(s) to create an execution cycle for

Module contents

ngclearn.generator package

Subpackages

ngclearn.generator.experimental package

Subpackages

ngclearn.generator.experimental.infimnist package

Submodules

ngclearn.generator.experimental.infimnist.infimnist module

Module contents

Module contents

ngclearn.generator.static package

Submodules

ngclearn.generator.static.mog module

`class ngclearn.generator.static.mog.MoG`(*x_dim=2*, *num_comp=1*, *means=None*, *covar=None*, *phi=None*, *assume_diag_cov=False*, *fscale=1.0*, *seed=69*)

Bases: `object`

Implements a mixture of Gaussians (MoG) stochastic data generating process.

Parameters

- **x_dim** – the dimensionality of the simulated data/input space
- **num_comp** – the number of components/latent variables within this GMM
- **assume_diag_cov** – if True, assumes a diagonal covariance for each component (Default = False)
- **means** – a list of means, each a (1 x D) vector (in tensor tf.float32 format)
- **covar** – a list of covariances, each a (D x D) vector (in tensor tf.float32 format)
- **assume_diag_cov** – if covar is None, forces auto-created covariance matrices to be strictly diagonal
- **fscale** – if covar is None, this controls the global scale of each component's covariance (Default = 1.0)
- **seed** – integer seed to control determinism of the underlying data generating process

`sample`(*n_s*, *mode_idx=- 1*)

Draw samples from the current underlying data generating mixture process.

Parameters

- **n_s** – the number of samples to draw from this GMM
- **mode_i** – if >= 0, will only draw samples from a selected, specific component of this process (Default = -1)

Module contents

ngclearn.generator.temporal package

Submodules

ngclearn.generator.temporal.noisy_sin module

class `ngclearn.generator.temporal.noisy_sin.`**`NoisySinusoid`**(*sigma*, *dt=0.01*, *x_initial=None*)

Bases: `object`

Implements the noisy sinusoid stochastic (temporal) data generating process. Note that centered Gaussian noise is used to create the corrupted samples of the underlying sinusoidal process.

Parameters

- **sigma** – a (1 x D) vector (numpy) that dictates the standard deviation of the process
- **dt** – the integration time step (Default = 0.01)
- **x_initial** – the initial value of the process (Default = None, yielding a zero vector starting point)

reset()

Resets the temporal noise process back to its initial condition/starting point.

sample()

Draws a sample from the current state of this noisy sinusoidal temporal process.

Returns a vector sample of shape (1 x D)

ngclearn.generator.temporal.oh_process module

class `ngclearn.generator.temporal.oh_process.`**`OUNoise`**(*mean*, *std_deviation*, *theta=0.15*, *dt=0.01*, *x_initial=None*)

Bases: `object`

Implements an Ornstein Uhlenbeck (O-H) stochastic (temporal) data generating process.

Parameters

- **mean** – a (1 x D) vector (numpy) (mean of the process)
- **std_deviation** – a (1 x D) vector (numpy) (standard deviation of the process)
- **theta** – meta-parameter to control the drift term of the process (Default = 0.15)
- **dt** – the integration time step (Default = 1e-2)
- **x_initial** – the initial value of the process (Default = None, yielding a zero vector starting point)

reset()

Resets the temporal noise process back to its initial condition/starting point.

sample()

Draws a sample from the current state of this O-H temporal process.

Returns a vector sample of shape (1 x D)

Module contents

Module contents

ngclearn.museum package

Submodules

ngclearn.museum.gncn_pdh module

class ngclearn.museum.gncn_pdh.**GNCN_PDH**(*args*)

Bases: `object`

Structure for constructing the model proposed in:

Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022).

This model, under the NGC computational framework, is referred to as the GNCN-t1-Sigma/Friston, according to the naming convention in (Ororbia & Kifer 2022).

Historical Note:
(The arXiv paper that preceded the publication above is shown below:)
Ororbia, Alexander, and Daniel Kifer. "The neural coding framework for
learning generative models." arXiv preprint arXiv:2012.03405 (2020).

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
z3 -(z3-mu1)-> mu1; z2 -(z2-mu0)-> mu0
e2 -> e2 * Sigma2; e1 -> e1 * Sigma1 // Precision weighting
z3 -> z3 * Lat3; z2 -> z2 * Lat2; z1 -> z1 * Lat1 // Lateral competition
e2 -(e2-z3)-> z3; e1 -(e1-z2)-> z2; e0 -(e0-z1)-> z1 // Error feedback

Parameters **`args`** – a Config dictionary containing necessary meta-parameters for the GNCN-PDH

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim: # of latent variables in layer z3 (top-most layer)
* z_dim: # of latent variables in layers z1 and z2
* x_dim: # of latent variables in layer z0 or sensory x
* seed: number to control determinism of weight initialization
* wght_sd: standard deviation of Gaussian initialization of weights
* beta: latent state update factor
* leak: strength of the leak variable in the latent states
* K: # of steps to take when conducting iterative inference/settling
* act_fx: activation function for layers z1, z2, and z3

* out_fx: activation function for layer mu0 (prediction of z0) (Default: sigmoid)
* n_group: number of neurons w/in a competition group for z2 and z2 (sizes of z2 and z1 should be divisible by this number)
* n_top_group: number of neurons w/in a competition group for z3 (size of z3 should be divisible by this number)
* alpha_scale: the strength of self-excitation
* beta_scale: the strength of cross-inhibition

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

`print_norms`()

Prints the Frobenius norms of each parameter of this system

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters **`z_sample`** – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`set_weights`(*source*, *tau=0.005*)

Deep copies weight variables of another model (of the same exact type) into this model's weight variables/parameters.

Parameters

- **`source`** – the source model to extract/transfer params from
- **`tau`** – if > 0, the Polyak averaging coefficient (-1 sets to hard deep copy/transfer)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- **`x`** – sensory input to reconstruct/predict
- **`calc_update`** – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`update`(*x*, *avg_update=True*)

Updates synaptic parameters/connections given inputs x and y

Parameters **`x`** – a sensory sample or batch of sensory samples

ngclearn.museum.gncn_t1 module

class `ngclearn.museum.gncn_t1.`**`GNCN_t1`**`(`*args*`)`

Bases: `object`

Structure for constructing the model proposed in:

Rao, Rajesh PN, and Dana H. Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects." Nature neuroscience 2.1 (1999): 79-87.

Note this model includes the Laplacian prior to induce some level of sparsity in the latent activities. This model, under the NGC computational framework, is referred to as the GNCN-t1/Rao, according to the naming convention in (Ororbia & Kifer 2022).

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0

Parameters **`args`** – a Config dictionary containing necessary meta-parameters for the GNCN-t1

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim - # of latent variables in layer z3 (top-most layer)
* z_dim - # of latent variables in layers z1 and z2
* x_dim - # of latent variables in layer z0 or sensory x
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* beta - latent state update factor
* leak - strength of the leak variable in the latent states
* lmbda - strength of the Laplacian prior applied over latent state activities
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1, z2, and z3
* out_fx - activation function for layer mu0 (prediction of z0) (Default: sigmoid)

`calc_updates``(`*avg_update=True*, *decay_rate=- 1.0*`)`

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear``()`

Clears the states/values of the stateful nodes in this NGC system

`print_norms``()`

Prints the Frobenius norms of each parameter of this system

`project``(`*z_sample*`)`

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters **`z_sample`** – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`set_weights`(*source*, *tau=0.005*)

Deep copies weight variables of another model (of the same exact type) into this model's weight variables/parameters.

Parameters

- `source` – the source model to extract/transfer params from
- `tau` – if > 0, the Polyak averaging coefficient (-1 sets to hard deep copy/transfer)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`update`(*x*, *avg_update=True*)

Updates synaptic parameters/connections given inputs x and y

Parameters `x` – a sensory sample or batch of sensory samples

ngclearn.museum.gncn_t1_ffm module

class `ngclearn.museum.gncn_t1_ffm.GNCN_t1_FFM`(*args*)

Bases: `object`

Structure for constructing the model proposed in:

Whittington, James CR, and Rafal Bogacz. "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity." Neural computation 29.5 (2017): 1229-1262.

This model, under the NGC computational framework, is referred to as the GNCN-t1-FFM, a slightly modified from of the naming convention in (Ororbia & Kifer 2022, Supplementary Material). "FFM" denotes feedforward mapping.

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
Note that z3 = x and z0 = y, yielding a classifier or regressor

Parameters `args` – a Config dictionary containing necessary meta-parameters for the GNCN-t1-FFM

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* x_dim - # of latent variables in layer z3 or sensory input x
* z_dim - # of latent variables in layers z1 and z2

* y_dim - # of latent variables in layer z0 or output target y
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* beta - latent state update factor
* leak - strength of the leak variable in the latent states
* lmbda - strength of the Laplacian prior applied over latent state activities
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1, z2
* out_fx - activation function for layer mu0 (prediction of z0 or y) (Default: identity)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

`predict`(*x*)

Predicts the target (either a probability distribution over labels, i.e., p(y|x), or a vector of regression targets) for a given *x*

Parameters `z_sample` – the input sample to project through the NGC graph

Returns y_sample (sample(s) of the underlying predictive model)

`print_norms`()

Prints the Frobenius norms of each parameter of this system

`project`(*x_sample*)

(Internal function)
Run projection scheme to get a sample of the underlying directed
generative model given the clamped variable *z_sample = x*

Parameters `x_sample` – the input sample to project through the NGC graph

Returns y_sample (sample(s) of the underlying predictive model)

`set_weights`(*source*, *tau=- 1.0*)

Deep copies weight variables of another model (of the same exact type) into this model's weight variables/parameters.

Parameters

- `source` – the source model to extract/transfer params from
- `tau` – if > 0, the Polyak averaging coefficient (-1 sets to hard deep copy/transfer)

`settle`(*x*, *y*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to clamp top-most layer (z3) to
- `y` – target output activity, i.e., label or regression target

- **calc_update** – if True, computes synaptic updates @ end of settling process (Default = True)

Returns y_hat (predicted y)

update(*x*, *y*, *avg_update=True*)

Updates synaptic parameters/connections given inputs x and y

Parameters

- **x** – a sensory sample or batch of sensory samples
- **y** – a target or batch of targets

ngclearn.museum.gncn_t1_sc module

class ngclearn.museum.gncn_t1_sc.**GNCN_t1_SC**(*args*)

Bases: `object`

Structure for constructing the sparse coding model proposed in:

Olshausen, B., Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996).

Note this model imposes a factorial (Cauchy) prior to induce sparsity in the latent activities z1 (the latent codebook). Synapses initialized from a (fan-in) scaled uniform distribution. This model would be named, under the NGC computational framework naming convention (Ororbia & Kifer 2022), as the GNCN-t1/SC (SC = sparse coding) or GNCN-t1/Olshausen.

Node Name Structure:
p(z1) ; z1 -(z1-mu0)-> mu0 ;e0; z0
Cauchy prior applied for p(z1)

Note: You can also recover the model learned through ISTA by using, instead of a factorial prior over latents, a thresholding function such as the "soft_threshold". (Make sure you set "prior" to "none" in this case.) This results in the GNCN-t1/SC emulating a system similar to that proposed in:

Daubechies, Ingrid, Michel Defrise, and Christine De Mol. "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint." Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences 57.11 (2004): 1413-1457.

Parameters **args** – a Config dictionary containing necessary meta-parameters for the GNCN-t1/SC

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_dim - # of latent variables in layers z1
* x_dim - # of latent variables in layer z0 or sensory x
* seed - number to control determinism of weight initialization
* beta - latent state update factor
* leak - strength of the leak variable in the latent states (Default = 0)
* prior - type of prior to use (Default = "cauchy")

* lmbda - strength of the prior applied over latent state activities (only if prior != "none")
* threshold - type of threshold to use (Default = "none")
* thr_lmbda - strength of the threshold applied over latent state activities (only if threshold != "none")
* n_group - must be > 0 if lat_type != None and s.t. (z_dim mod n_group) == 0
* K - # of steps to take when conducting iterative inference/settling
* act_fx - activation function for layers z1 (Default = identity)
* out_fx - activation function for layer mu0 (prediction of z0) (Default: identity)

`calc_updates`(*avg_update=True*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

`print_norms`()

Prints the Frobenius norms of each parameter of this system

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters `z_sample` – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`set_weights`(*source*, *tau=0.005*)

Deep copies weight variables of another model (of the same exact type) into this model's weight variables/parameters.

Parameters

- `source` – the source model to extract/transfer params from
- `tau` – if > 0, the Polyak averaging coefficient (-1 sets to hard deep copy/transfer)

`settle`(*x*, *K=- 1*, *cold_start=True*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `K` – number of steps to run iterative settling for
- `cold_start` – start settling process states from zero (Leave this to True)
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`update`(*x*, *avg_update=True*)

Updates synaptic parameters/connections given sensory input

Parameters `x` – a sensory sample or batch of sensory samples

ngclearn.museum.gncn_t1_sigma module

class `ngclearn.museum.gncn_t1_sigma.`**`GNCN_t1_Sigma`**(*args*)

Bases: `object`

Structure for constructing the model proposed in:

Friston, Karl. "Hierarchical models in the brain." PLoS Computational Biology 4.11 (2008): e1000211.

Note this model includes a Laplacian prior to induce some level of sparsity in the latent activities. This model, under the NGC computational framework, is referred to as the GNCN-t1-Sigma/Friston, according to the naming convention in (Ororbia & Kifer 2022).

Node Name Structure:
z3 -(z3-mu2)-> mu2 ;e2; z2 -(z2-mu1)-> mu1 ;e1; z1 -(z1-mu0-)-> mu0 ;e0; z0
e2 -> e2 * Sigma2; e1 -> e1 * Sigma1 // Precision weighting

Parameters **`args`** – a Config dictionary containing necessary meta-parameters for the GNCN-t1-Sigma

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_top_dim: # of latent variables in layer z3 (top-most layer)
* z_dim: # of latent variables in layers z1 and z2
* x_dim: # of latent variables in layer z0 or sensory x
* seed: number to control determinism of weight initialization
* wght_sd: standard deviation of Gaussian initialization of weights
* beta: latent state update factor
* leak: strength of the leak variable in the latent states
* lmbda: strength of the Laplacian prior applied over latent state activities
* K: # of steps to take when conducting iterative inference/settling
* act_fx: activation function for layers z1, z2, and z3
* out_fx: activation function for layer mu0 (prediction of z0) (Default: sigmoid)

`calc_updates`(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic matrix updates (that follow order of .theta)

`clear`()

Clears the states/values of the stateful nodes in this NGC system

`print_norms`()

Prints the Frobenius norms of each parameter of this system

`project`(*z_sample*)

Run projection scheme to get a sample of the underlying directed generative model given the clamped variable *z_sample*

Parameters **`z_sample`** – the input noise sample to project through the NGC graph

Returns x_sample (sample(s) of the underlying generative model)

`set_weights`(*source*, *tau=0.005*)

Deep copies weight variables of another model (of the same exact type) into this model's weight variables/parameters.

Parameters

- `source` – the source model to extract/transfer params from
- `tau` – if > 0, the Polyak averaging coefficient (-1 sets to hard deep copy/transfer)

`settle`(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables

Parameters

- `x` – sensory input to reconstruct/predict
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns x_hat (predicted x)

`update`(*x*, *avg_update=True*)

Updates synaptic parameters/connections given inputs x and y

Parameters `x` – a sensory sample or batch of sensory samples

ngclearn.museum.harmonium module

`class ngclearn.museum.harmonium.Harmonium`(*args*)

Bases: `object`

Structure for constructing the Harmonium model proposed in:

Hinton, Geoffrey E. "Training products of experts by maximizing contrastive likelihood." Technical Report, Gatsby computational neuroscience unit (1999).

Node Name Structure:
z1 -(z1-z0)-> z0
z0 -(z0-z1)-> z1
Note: z1-z0 = (z0-z1)^T (transpose-tied synapses)

Another important reference for designing stable Harmoniums is here:

Hinton, Geoffrey E. "A practical guide to training restricted Boltzmann machines." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 599-619.

Note: if you set the *samp_fx* to the "identity", you force the Harmonium to to work as a mean-field Harmonium/Botlzmann machine

Parameters `args` – a Config dictionary containing necessary meta-parameters for the Harmonium

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model

* z_dim - # of latent variables in layer z1
* x_dim - # of latent variables in layer z0 (or sensory x)
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights
* K - # of steps to take when conducting Contrastive Divergence
* act_fx - activation function for layer z1 (Default: sigmoid)
* out_fx - activation function for layer z0 (prediction of z0) (Default: sigmoid)
* samp_fx - sampling function for layer z1 (Default = bernoulli)

calc_updates(*avg_update=True*, *decay_rate=- 1.0*)

Calculate adjustments to parameters under this given model and its current internal state values

Returns delta, a list of synaptic updates (that follow order of pos_phase.theta)

clear()

Clears the states/values of the stateful nodes in this NGC system

print_norms()

Prints the Frobenius norms of each parameter of this system

sample(*K*, *x_sample=None*, *batch_size=1*)

Samples the underlying harmonium to generate a chain of patterns from a block Gibbs sampling process.

Parameters

- **K** – number of steps to run the Gibbs sampler
- **x_sample** – inital condition for the sampler (Default = None), if None, this will generate an initial sample of size (batch_size, z1_dim) where z1_dim is the dimensionality of the latent state.
- **batch_size** – if x_sample is None, then this dictates how many samples in parallel to create per step of running the Gibbs sampler

settle(*x*, *calc_update=True*)

Run an iterative settling process to find latent states given clamped input and output variables.

Parameters

- **x** – sensory input to reconstruct/predict
- **calc_update** – if True, computes synaptic updates @ end of settling process for both NGC system and inference co-model (Default = True)

Returns x_hat (predicted x)

ngclearn.museum.snn_ba module

class ngclearn.museum.snn_ba.**SNN_BA**(*args*)

Bases: `object`

A spiking neural network (SNN) classifier that adapts its synaptic cables via broadcast alignment. Specifically, this model is a generalization of the one proposed in:

Samadi, Arash, Timothy P. Lillicrap, and Douglas B. Tweed. "Deep learning with dynamic spiking neurons and fixed feedback weights." Neural computation 29.3 (2017): 578-602.

This model encodes its real-valued inputs as Poisson spike trains with spikes emitted at a rate of approximately 63.75 Hz. The internal nodes and output nodes operate under the leaky integrate-and-fire spike response model and operate with a relative refractory rate of 1.0 ms. The integration time constant for this model has been set to 0.25 ms.

Node Name Structure:
z2 -(z2-mu1)-> mu1 ; z1 -(z1-mu0-)-> mu0 ;e0; z0
e0 -> d1 and z1 -> d1, where d1 is a teaching signal for z1
  Note that z2 = x and z0 = y, yielding a classifier

Parameters `args` – a Config dictionary containing necessary meta-parameters for the SNN-BA

DEFINITION NOTE:
args should contain values for the following:
* batch_size - the fixed batch-size to be fed into this model
* z_dim - # of latent variables in layers z1
* x_dim - # of latent variables in layer z2 or sensory x
* y_dim - # of variables in layer z0 or target y
* seed - number to control determinism of weight initialization
* wght_sd - standard deviation of Gaussian initialization of weights (optional)
* T - # of time steps to take when conducting iterative settling (if not online)

`clear()`

Clears the states/values of the stateful nodes in this NGC system

`predict(x)`

Predicts the target for a given *x*. Specifically, this function will return spike counts, one per class in *y* – taking the argmax of these counts will yield the model's predicted label.

Parameters `z_sample` – the input sample to project through the NGC graph

Returns y_sample (spike counts from the underlying predictive model)

`settle(x, y=None, calc_update=True)`

Run an iterative settling process to find latent states given clamped input and output variables, specifically simulating the dynamics of the spiking neurons internal to this SNN model. Note that this functions returns two outputs – the first is a count matrix (each row is a sample in mini-batch) and each column represents the count for one class in y, and the second is an approximate probability distribution computed as a softmax over an average across the electrical currents produced at each step of simulation.

Parameters

- `x` – sensory input to clamp top-most layer (z2) to
- `y` – target output activity, i.e., label target
- `calc_update` – if True, computes synaptic updates @ end of settling process (Default = True)

Returns

y_count (spike counts per class in y), y_hat (approximate probability distribution for y)

Module contents

ngclearn.utils package

Subpackages

ngclearn.utils.experimental package

Submodules

ngclearn.utils.experimental.viz_graph_utils module

Module contents

Submodules

ngclearn.utils.config module

`class ngclearn.utils.config.`**`Config`**`(`*fname=None*`)`

Bases: `object`

Simple configuration object to house named arguments for experiments (to be built from a .cfg file on disk).

File format is:
# Comments start with pound symbol
arg_name = arg_value
arg_name = arg_value # side comment that will be stripped off

Parameters **`fname`** – source file name to build configuration object from (suffix = .cfg)

`getArg``(`*arg_name*`)`

Retrieve argument from current configuration

Parameters **`arg_name`** – the string name of the argument to retrieve from this config

Returns the value of the named argument queried

`hasArg``(`*arg_name*`)`

Check if argument exists (or if it is known by this config object)

Parameters **`arg_name`** – the string name of the argument to check for the existence of

Returns True if this config contains this argument, False otherwise

`setArg``(`*arg_name*, *arg_value*`)`

Sets an argument directly

Parameters

- **`arg_name`** – the string name of the argument to set within this config
- **`arg_value`** – the value name of the argument to set within this config

ngclearn.utils.data_utils module

Data functions and utilies.

class ngclearn.utils.data_utils.**DataLoader**(*design_matrices*, *batch_size*, *disable_shuffle=False*, *ensure_equal_batches=True*)

Bases: `object`

A data loader object, meant to allow sampling w/o replacement of one or more named design matrices. Note that this object is iterable (and implements an __iter__() method).

Parameters

- **design_matrices** – list of named data design matrices - [("name", matrix), . . .]
- **batch_size** – number of samples to place inside a mini-batch
- **disable_shuffle** – if True, turns off sample shuffling (thus no sampling w/o replacement)
- **ensure_equal_batches** – if True, ensures sampled batches are equal in size (Default = True). Note that this means the very last batch, if it's not the same size as the rest, will reuse random samples from previously seen batches (yielding a batch with a mix of vectors sampled with and without replacement).

ngclearn.utils.data_utils.**binarized_shuffled_omniglot**(*out_dir*)

Specialized function for the omniglot dataset.

Note: this function has not been tested/fully integrated yet

ngclearn.utils.data_utils.**generate_patch_set**(*imgs*, *patch_shape*, *batch_size*, *center_patch=True*)

Generates a set of patches from an array/list of image arrays (via random sampling with replacement).

Parameters

- **imgs** – the array of image arrays to sample from
- **patch_shape** – a 2-tuple of the form (pH = patch height, pW = patch width)
- **batch_size** – how many patches to extract/generate from source images
- **center_patch** – centers each patch by subtracting the patch mean (per-patch)

Returns an array (D x (pH * pW)), where each row is a flattened patch sample

ngclearn.utils.io_utils module

Utility I/O functions.

ngclearn.utils.io_utils.**deserialize**(*fname*)

De-serializes a object from disk

Parameters **fname** – filename of object to load - /path/to/fname_of_model

Returns the deserialized model object

ngclearn.utils.io_utils.**parse_simulation_info**(*sim_info*)

Parses a simulation information dictionary into a human-readable string.

Parameters **sim_info** – simulation info dictionary

Returns a string presenting the simulation information

ngclearn.utils.io_utils.**plot_img_grid**(*samples*, *fname*, *nx*, *ny*, *px*, *py*, *plt*, *rotNeg90=False*) ngclearn.utils.io_utils.**plot_sample_img**(*x_s*, *px*, *py*, *fname*, *plt*, *rotNeg90=False*)

Plots a (1 x (px * py)) array as a (px x py) gray-scale image and saves this image to disk.

Parameters

- **x_s** – the numpy image array
- **px** – number of pixels in the row dimension
- **py** – number of pixels in the column dimension
- **fname** – the filename of the image to save to disk
- **plt** – a matplotlib plotter object
- **rotNeg90** – rotates the image -90 degrees before saving to disk

ngclearn.utils.io_utils.**serialize**(*fname*, *object*)

Serializes an object to disk.

Parameters

- **fname** – filename of object to save - /path/to/fname_of_model
- **model** – model object to serialize

ngclearn.utils.metric_utils module

General mathematical measurement/metric functions/utilities file.

ngclearn.utils.metric_utils.**bce**(*p*, *x*, *offset=1e-07*)

Calculates the negative Bernoulli log likelihood or binary cross entropy (BCE).

Parameters

- **p** – predicted probabilities of shape (N x D)
- **x** – target binary values (data) of shape (N x D)

Returns an (N x 1) column vector, where each row is the BCE(p, x) for that row's datapoint

ngclearn.utils.metric_utils.**calc_ACC**(*T*)

Calculates the average accuracy (ACC) given a task matrix T.

Parameters **T** – task matrix (containing accuracy values)

Returns scalar ACC for T

ngclearn.utils.metric_utils.**calc_BWT**(*T*)

Calculates the backward(s) transfer (BWT) given a task matrix T

Parameters **T** – task matrix (containing accuracy values)

Returns scalar BWT for T

ngclearn.utils.metric_utils.**cat_nll**(*p*, *x*, *epsilon=1e-07*)

Measures the negative Categorical log likelihood

Parameters

- **p** – predicted probabilities
- **x** – true one-hot encoded targets

Returns an (N x 1) column vector, where each row is the Cat.NLL(x_pred, x_true) for that row's datapoint ngclearn.utils.metric_utils.**fast_log_loss**(*probs*, *y_ind*)

Calculates negative Categorical log likelihood / cross entropy via a fast indexing approach (assumes targets/labels are integers or class indices for single-class one-hot encoding).

Parameters

- **probs** – predicted label probability distributions (one row per label)
- **y_ind** – label indices - can be either (D,) vector or (Dx1) column vector

Returns the scalar value of Cat.NLL(x_pred, x_true)

ngclearn.utils.metric_utils.**mse**(*mu*, *x*)

Measures mean squared error (MSE), or the negative Gaussian log likelihood with variance of 1.0.

Parameters

- **mu** – predicted values (mean)
- **x** – target values (x/data)

Returns an (N x 1) column vector, where each row is the MSE(x_pred, x_true) for that row's datapoint

ngclearn.utils.stat_utils module

Statistical functions/utilities file.

ngclearn.utils.stat_utils.**ainv**(*A*)

Computes the inverse of matrix A

Parameters **A** – matrix to invert

Returns the inversion of A

ngclearn.utils.stat_utils.**calc_covariance**(*X*, *mu_=None*, *weights=None*, *bias=True*)

Calculate the covariance matrix of X

Parameters

- **X** – an (N x D) data design matrix to measure log density over (1 row vector - 1 data point)
- **mu** – a pre-computed (1 x D) vector mean of the Gaussian distribution (Default = None)
- **weights** – a (N x 1) weighting column vector, one row is weight applied to one sample in X (Default = None)
- **bias** – (only applies if weights is None), if True, compute the biased estimator of covariance

Returns a (D x D) covariance matrix

ngclearn.utils.stat_utils.**calc_gKL**(*mu_p*, *sigma_p*, *mu_q*, *sigma_q*)

Calculate the Gaussian Kullback-Leibler (KL) divergence between two multivariate Gaussian distributions, i.e., KL(p||q).

Parameters

- **mu_p** – (1 x D) vector mean of distribution p
- **sigma_p** – (D x D) covariance matrix of distributon p
- **mu_q** – (1 x D) vector mean of distribution q
- **sigma_q** – (D x D) covariance matrix of distributon q

Returns the scalar KL divergence

`ngclearn.utils.stat_utils.calc_list_moments(data_list, num_dec=3)`

Compute the mean and standard deviation from a list of data values. This is for simple scalar measurements/metrics that will be printed to I/O.

Parameters

- `data_list` – list of data values, each element should be (1 x 1)
- `num_dec` – number of decimal points to round values to (Default = 3)

Returns (mu, sigma), where mu = mean and sigma = standard deviation

`ngclearn.utils.stat_utils.calc_log_gauss_pdf(X, mu, cov)`

Calculates the log Gaussian probability density function (PDF)

Parameters

- `X` – an (N x D) data design matrix to measure log density over
- `mu` – the (1 x D) vector mean of the Gaussian distribution
- `cov` – the (D x D) covariance matrix of the Gaussian distribution

Returns a (N x 1) column vector w/ each row containing log density value per sample

`ngclearn.utils.stat_utils.convert_to_spikes(x_data, gain=1.0, offset=0.0, n_trials=1)`

`ngclearn.utils.stat_utils.sample_bernoulli(p)`

Samples a multivariate Bernoulli distribution

Parameters `p` – probabilities to samples of shape (n_s x D)

Returns an (n_s x D) (binary) matrix of Bernoulli samples (one vector sample per row)

`ngclearn.utils.stat_utils.sample_gaussian(n_s, mu=0.0, sig=1.0, n_dim=- 1)`

Samples a multivariate Gaussian assuming a diagonal covariance or scalar variance (shared across dimensions) in the form of a standard deviation vector/scalar.

Parameters

- `n_s` – number of samples to draw
- `mu` – (1 x D) mean of the Gaussian distribution
- `sig` – (1 x D) or (1 x 1) standard deviation of the Gaussian distribution
- `n_dim` – dimensionality of the sample space

Returns an (n_s x n_dim) matrix of uniform samples (one vector sample per row)

`ngclearn.utils.stat_utils.sample_uniform(n_s, n_dim)`

Samples a multivariate Uniform distribution

Parameters

- `n_s` – number of samples to draw
- `n_dim` – dimensionality of the sample space

Returns an (n_s x n_dim) matrix of uniform samples (one vector sample per row)

ngclearn.utils.transform_utils module

A mathematical transformation utilities function file. This file contains activation functions and other relevant data transformation tools/utilities.

`ngclearn.utils.transform_utils.`**`bernoulli`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`binarize`**`(`*data*, *threshold=0.5*`)`

Converts the vector *data* to its binary equivalent

Parameters

- **`data`** – the data to binarize (real-valued)
- **`threshold`** – the cut-off point for 0, i.e., if threshold = 0.5, then any number/value inside of data < 0.5 is set to 0, otherwise, it is set to 1.0

Returns the binarized equivalent of "data"

`ngclearn.utils.transform_utils.`**`binary_flip`**`(`*x_b*`)`

Flips the bit values within binary vector *x_b*

Parameters **`x_b`** – the binary vector to flip

Returns the flipped binary vector form of x_b

`ngclearn.utils.transform_utils.`**`bkwta`**`(`*x*, *K=10*`)`

Binarized k-winners-take-all competitive activation function

`ngclearn.utils.transform_utils.`**`calc_modulatory_factor`**`(`*W*`)`

Calculate modulatory matrix W_M for W

Note: this is NOT fully tested/integrated yet

`ngclearn.utils.transform_utils.`**`calc_zca_whitening_matrix`**`(`*X*`)`

Calculates a ZCA whitening matrix via the Mahalanobis whitening method.

Note: this is NOT fully tested/integrated yet

Parameters **`X`** – a design matrix of shape (M x N), where rows -> features, columns -> observations

Returns the resultant (M x M) ZCA matrix

`ngclearn.utils.transform_utils.`**`clip_fx`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`convert_to_spikes_`**`(`*x_data*, *max_spike_rate*, *dt*, *sp_div=4.0*`)`

Converts a vector *x_data* to its approximate Poisson spike equivalent.

Note: this function is NOT fully tested/integrated yet.

Parameters

- **`max_spike_rate`** – firing rate (in Hertz)
- **`dt`** – integraton time constant (in milliseconds or ms)
- **`sp_div`** – to denominator to convert input data values to a firing frequency

Returns the binary spike vector form of *x_data*

`ngclearn.utils.transform_utils.`**`create_block_bin_matrix`**`(`*shape*, *n_ones_per_row*`)`

`ngclearn.utils.transform_utils.`**`create_competiion_matrix`**`(`*z_dim*, *lat_type*, *beta_scale*, *alpha_scale*, *n_group*, *band*`)`

This function creates a particular matrix to simulate competition via self-excitatory and inhibitory synaptic signals.

Parameters

- `z_dim` – dimensionality of neural group to apply competition to
- `lat_type` – type of competiton pattern. “lkwta” sets a column/group based form of k-WTA style competition and “band” sets a matrix band-based form of competition.
- `beta_scale` – the strength of the cross-unit inhibiton
- `alpha_scale` – the strength of the self-excitation
- `n_group` – if lat_type is set to “lkwta”, then this ensures that only a certain number of neurons are within a competitive group/column

  Note z_dim should be divisible by n_group
- `band` – the band parameter (Note: not fully tested)

Returns a (z_dim x z_dim) competition matrix

ngclearn.utils.transform_utils.**create_mask_matrix**(*n_col_m*, *nrow*, *ncol*)

ngclearn.utils.transform_utils.**d_elu**(*z*, *alpha=1.0*)

ngclearn.utils.transform_utils.**d_identity**(*x*)

ngclearn.utils.transform_utils.**d_ltanh**(*z*)

ngclearn.utils.transform_utils.**d_relu**(*x*)

ngclearn.utils.transform_utils.**d_relu6**(*x*)

ngclearn.utils.transform_utils.**d_sigmoid**(*x*)

ngclearn.utils.transform_utils.**d_softplus**(*x*)

ngclearn.utils.transform_utils.**d_tanh**(*x*)

ngclearn.utils.transform_utils.**decide_fun**(*fun_type*)

A selector function that generates a physical activation function and its first-order (element-wise) derivative funciton given a description *fun_type*. Note that some functions do not come with a proper derivative (and thus set to the identity function derivative – see list below).

Currently supported functions (for given fun_type) include:

* “tanh” - hyperbolic tangent
* “ltanh” - LeCun-style hyperbolic tangent
* “sigmoid” - logistic link function
* “kwta” - K-winners-take-all
* “softmax” - the softmax function (derivative not generated)
* “identity” - the identity function
* “relu” - rectified linear unit
* “lrelu” - leaky rectified linear unit
* “softplus” - the softplus function
* “relu6” - the relu but upper bounded/capped at 6.0
* “elu” - exponential linear unit
* “erf” - the error function (derivative not generated)
* “binary_flip” - bit-flipping function (derivative not generated)

* “bkwta” - binary K-winners-take-all (derivative not generated)
* “sign” - signum (derivative not generated)
* “clip_fx” - clipping function (derivative not generated)
* “heaviside” - Heaviside function (derivative not generated)
* “bernoulli” - the Bernoulli sampling function (derivative not generated)

Parameters `fun_type` – a string stating the name of activation function and its 1st elementwise derivative to generate

Returns (fx, d_fx), where fx is the physical activation function and d_fx its derivative

`ngclearn.utils.transform_utils.`**`drop_out`**(*input*, *rate=0.0*, *seed=69*)

Custom drop-out function – returns output as well as binary mask

`ngclearn.utils.transform_utils.`**`elu`**(*z*, *alpha=1.0*)

`ngclearn.utils.transform_utils.`**`erf`**(*x*)

`ngclearn.utils.transform_utils.`**`filter`**(*x_t*, *x_f*, *dt*, *a*, *filter_type='var_trace'*)

Applies a filter to data *x_t*.

Note: this function is NOT fully tested/integrated yet.

Parameters

- **`x_t`** –
- **`x_f`** –
- **`dt`** –
- **`a`** –
- **`filter_type`** – (Default = “var_trace”)

Returns the filtered vector form of x_t

`ngclearn.utils.transform_utils.`**`global_contrast_normalization`**(*Xin*, *s*, *lmda*, *epsilon*)

`ngclearn.utils.transform_utils.`**`gte`**(*x*, *val=0.0*)

`ngclearn.utils.transform_utils.`**`identity`**(*z*)

`ngclearn.utils.transform_utils.`**`init_weights`**(*kernel*, *shape*, *seed*)

Randomly generates/initializes a matrix/vector according to a kernel pattern.

Currently supported/tested patterns include:

* “he_uniform”
* “he_normal”
* “classic_glorot”
* “glorot_normal”
* “glorot_uniform”
* “orthogonal”
* “truncated_gaussian” (alternative: “truncated_normal”)
* “gaussian” (alternative: “normal”)
* “uniform”

Parameters

- `kernel` – a tuple denoting the pattern by which a matrix is initialized Note that the first item of *kernel* MUST contain a string specifying the initlialization pattern/scheme to use. Other elements, for tuples of length > 1 can contain pattern-specific hyper-paramters.
- `shape` – a 2-tuple specifying (N x M), a matrix of N rows by M columns
- `seed` – value to control determinism in initializer

Returns an (N x M) matrix randomly initialized to a chosen scheme

`ngclearn.utils.transform_utils.`**`inverse_logistic`**`(`*x*, *clip_bound=0.03*`)`

Inverse logistic link - logit function

`ngclearn.utils.transform_utils.`**`inverse_tanh`**`(`*x*`)`

Inverse hyperbolic tangent

`ngclearn.utils.transform_utils.`**`kwta`**`(`*x*, *K=50*`)`

k-winners-take-all competitive activation function

`ngclearn.utils.transform_utils.`**`lrelu`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`ltanh`**`(`*z*`)`

`ngclearn.utils.transform_utils.`**`mellowmax`**`(`*x*, *omega=1.0*, *axis=1*`)`

`ngclearn.utils.transform_utils.`**`mish`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`normalize_by_norm`**`(`*param*, *proj_mag=1.0*, *param_axis=0*`)`

`ngclearn.utils.transform_utils.`**`normalize_image`**`(`*image*`)`

Maps image array first to [0, image.max() - image.min()] then to [0, 1]
Arg: image: the image numpy.ndarray

Returns image array mapped to [0, 1]

`ngclearn.utils.transform_utils.`**`relu`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`relu6`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`scale_feat`**`(`*x*, *a=- 1.0*, *b=1.0*`)`

Applies the min-max feature scaling function to input x.

Parameters

- `a` – the lower bound to scale *x* w/in
- `b` – the upper bound to scale *x* w/in

Returns the scaled version of *x*, w/ each value in range [a,b]

`ngclearn.utils.transform_utils.`**`sech`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`sech_sqr`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`shrink`**`(`*a*, *b*`)`

`ngclearn.utils.transform_utils.`**`sigmoid`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`sign`**`(`*x*`)` `ngclearn.utils.transform_utils.`**`softmax`**`(`*x*, *tau=0.0*`)`

Softmax function with overflow control built in directly. Contains optional temperature parameter to control sharpness (tau > 1 softens probs, < 1 sharpens –> 0 yields point-mass)

Parameters

- **`x`** – a (N x D) input argument (pre-activity) to the softmax operator
- **`tau`** – probability sharpening/softening factor

Returns a (N x D) probability distribution output block

`ngclearn.utils.transform_utils.`**`softplus`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`tanh`**`(`*x*`)`

`ngclearn.utils.transform_utils.`**`threshold_cauchy`**`(`*x*, *lmda*`)`

`ngclearn.utils.transform_utils.`**`threshold_soft`**`(`*x*, *lmda*`)`

`ngclearn.utils.transform_utils.`**`to_one_hot`**`(`*idx*, *depth*`)`

Converts an integer or integer array into a binary one-hot encoding.

Parameters

- **`idx`** – an integer or integer list representing the index/indices of the chosen category/categories
- **`depth`** – total number of actual categories (the dimension K of the encoding)

Returns a binary one-of-K encoding of the input idx (an N x K vector if len(idx) = N)

`ngclearn.utils.transform_utils.`**`whiten`**`(`*X*`)`

Whitens image X via ZCA whitening

Note: this is NOT fully tested/integrated yet

Module contents

23.1.2 Module contents

CHAPTER TWENTYFOUR

LIST OF PAPERS/PUBLICATIONS

The following is a list of current papers that use ngc-learn (this list will be actively updated as we discover others that use ngc-learn):

1. Ororbia, A., and Kifer, D. The neural coding framework for learning generative models. Nature Communications 13, 2064 (2022).
2. Ororbia, A., and Mali, A. Backprop-free reinforcement learning with active neural generative coding. Proceedings of the aaai conference on artificial intelligence (2022).
3. Ororbia, A. "Spiking neural predictive coding for continual learning from data streams." arXiv preprint arXiv:1908.08655 (2019).
4. Ororbia, A, and Kelly, M. Alex. "CogNGen: Constructing the Kernel of a Hyperdimensional Predictive Processing Cognitive Architecture." arXiv preprint arXiv:2204.00619 (2022).

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX