Programming Spiking Neural Networks on Intel Loihi

Chit-Kwan Lin, Andreas Wild, Gautham N. Chinya, Yongqiang Cao,
Mike Davies, Narayan Srinivasa, Daniel M. Lavery, Hong Wang
Intel Labs
chit-kwan.lin@intel.com

Abstract

Loihi is Intel's novel, many-core neuromorphic processor and is the first of its kind to feature a programmable learning engine that enables on-chip training of spiking neural networks (SNNs). Here, we present the Loihi toolchain, which consists of an intuitive Python-based API for specifying SNNs, a compiler and runtime for building and executing SNNs on Loihi, and several target platforms (Loihi silicon, FPGA, and functional simulator). To showcase the toolchain, we walk through how to build, train, and use a SNN to classify handwritten digits from the MNIST database.

Keywords neuromorphic computing, programming paradigms

1 Introduction

Spiking neural networks (SNNs) represent a new model of computing—one that casts computation as the time-dependent, state evolution of a dynamical system, composed of many parallel computing elements (neurons). These elements have local memory, exhibit local state dynamics, and communicate solely via spike impulses. Taking cues from biology, the discipline of neuromorphic computing attempts to adopt the brain's locality, fine-grain parallelism, and event-driven operation in building highly-efficient and scalable computing machines, by realizing SNNs in dedicated hardware. To date, there have been a number of efforts to build large-scale neuromorphic systems–ranging from analog designs, such as HICANN [1] and NeuroGrid [2], to digital ones, such as SpiNNaker [3] and IBM's TrueNorth [4] processor—all exploring different architectures to accelerate and scale up the computation of SNNs.

Recently, Intel unveiled *Loihi* [5], our state-of-the-art, fully digital, asynchronous, neuromorphic processor. Loihi features a many-core mesh comprised of 128 neuromorphic cores (hereafter, "neuro cores"); three embedded IA cores for managing the neuro cores and controlling spike traffic into and out of the chip; and an asynchronous network-on-chip (NoC) for transporting messages between cores. Most importantly, it is the first of its kind to feature *on-chip learning* via a micro-code-based learning rule engine within each neuro core. As a result, this novel system requires a new means of programming, beyond what current end-to-end software frameworks for neuromorphic hardware, such as PyNN[6], Nengo [7], and TrueNorth Corelets[8], support.

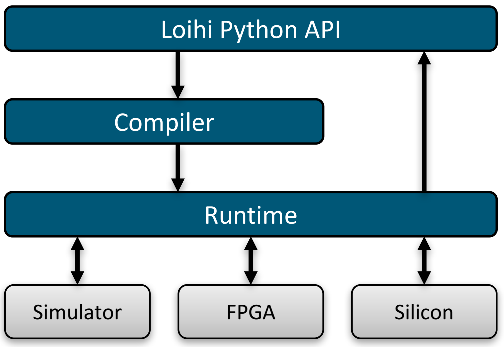

Figure 1. The Loihi toolchain consists of a Python API, a compiler and runtime, and several backend targets, namely, a functional simulator, a FPGA emulator, and the Loihi silicon.

In this paper, we introduce our toolchain (Figure 1) to configure SNNs for Loihi. We present a Python-based API that can be used to specify complex SNN topologies and to program custom learning rules; a compiler and runtime library for building and executing SNNs on Loihi; and, three backend targets—Loihi silicon, FPGA emulator, and software functional simulator.

To showcase the toolchain, we provide an end-to-end example of a single-layer SNN that is capable of learning and classifying images of handwritten digits from the well-known MNIST database [9]. To acquaint readers to this new model of computing, we will walk through the example step-by-step: from the encoding of the input images into spikes patterns, to the design of the SNN topology and supervised learning rules, to their implementation in our API, and to extracting and interpreting the classifier's results.

It is our hope that this toolchain will ease the task of programming SNNs on Loihi and, as a result, be a catalyst for greater participation in neuromorphic computing from the broader research community.

2 Programming Model

Existing SNN frameworks [6, 7, 10, 11] all share similarities in the high-level abstractions they present to the programmer. The core primitives we have chosen for our programming model are neuronal *compartments* and synaptic connections (or *synapses*, for short) as a means of defining SNN topology; *synaptic traces* and a *neuron model* to describe SNN dynamics; and synaptic *learning rules*. We describe these in detail below.

2.1 SNN Topology

At the most basic level of abstraction, we describe our SNN as a weighted, directed graph $G(V, E)$ where the vertices V represent *compartments* and the weighted edges E represent *synapses.*

Our programming model also supports the more advanced abstraction of a *neuron*, which is a set of compartments organized in a tree topology (analogous to dendritic tree [12] structures in biology), with the compartment at the root being the only one permitted to emit spikes. All other compartments participating in the dendritic tree may output state values to parents and integrate such inputs from their children to update internal state. This mechanism provides a way to construct a rich set of non-linear neuronal input/output mappings.

2.2 SNN Dynamics

Both compartments and synapses maintain internal state and communicate information only via discrete-time spike impulses, explicitly incorporating the relative timing of such spike events into a computation. This necessitates some form of temporal information summarization to combine events from different times in a computation. We achieve this through the exponential smoothing of spike sequences, the results of which we call *traces.* For compartments, we maintain "conductance" and "potential" traces, while for synapses, we distinguish "pre-synaptic" (for spikes arriving at a synapse) and "post-synaptic" (for spikes emitted by a compartment) traces.

The specific way in which compartment and synapse states evolve over time and in response to spikes are governed by a *neuron model.* In our programming model, we assume a variant of the well-known current-based (CUBA) model [13], defined as a set of first-order differential equations in terms of the above traces and evaluated in discrete algorithmic time steps (the exact formulation of the model are beyond the scope of this paper).

2.3 Learning Rules

The essence of learning for a SNN is the adjustment of its synaptic state variables (e.g., weights) in response to input and output spike patterns. The specific way in which synaptic states are updated is governed by a *learning rule.* In our programming model, it is expressed as a finite-difference equation with respect to a synaptic state variable. The equation must follow a sum-of-products form:

$$Z_{i,j}(t) = Z_{i,j}(t-1) + \sum_{i} S \prod_{j} F_{i,j} \tag{1}$$

where Z is the synaptic state variable being updated; t is the discrete time step; i and j are compartment indices; S is a scaling constant; and F may be a synaptic state variable, a pre-synaptic trace, or a post-synaptic trace. While this sum-of-products constraint does limit the space of possible learning rules, we make this trade-off because it naturally translates into an efficient implementation in hardware.

3 Architectural Overview

The Loihi architecture supports the programming model we have just outlined above by providing hardware support for complex multi-compartment neurons; multi-variable synaptic connections; connection sharing (e.g., for efficiently representing convolutional network topologies); and a programmable learning engine offering different modes of on-chip learning, such as supervised, unsupervised, and reinforcement learning. The Loihi architecture implements this model as a fully digital, asynchronous, event-driven, many-core, compute-in-memory system that supports multi-chip scalability and utilizes barrier synchronization to deterministically advance algorithmic time. Individual neuro cores in a Loihi chip, are organized in an asynchronous NoC, which also includes several IA cores that are used to process spike I/O and manage system state.

Figure 2 illustrates Loihi's computational graph from two perspectives: a high-level view that shows its main logical entities (left) and another view of their major architectural counterparts (right).

Each box in the logical view of Figure 2 corresponds to an entity-set such as synapses (SYN_j) or compartments (C_l) and represents a table containing the various dynamic states and static configuration parameters per discrete entity.

- **Compartments:** Compartments (C_l) are the main building blocks used to configure simple, single-compartment neurons as well as complex, multi-compartment neurons. Both integrate incoming spikes as well as input from other compartments. Compartments at the root of a dendritic tree produce a spike when they cross an activation threshold. When learning is enabled, spikes may propagate backwards to update post-synaptic spike traces ($Y_{l,p}$). For each such compartment there may be many output axons ($A_i^{(out)}$) that deliver spikes to their destinations, reminiscent of multiple axonal arbors in biology.
- **Axons:** Axons connect neurons to synapses. Here, we distinguish between output and input axons ($A_i^{(out)}$ and $A_i^{(in)}$). In our model, spikes arrive at input axons either from within the network or from external sources of stimulus. An input axon indexes a variable-length list of synapses (SYN_j) and, when learning is enabled, multiple synaptic pre-traces ($X_{i,p}$). Note that the axon construct is not present in the programming model and is introduced here at a lower level of abstraction because it is particular to the hardware implementation.
- **Synaptic traces:** There are multiple pre- ($X_{i,p}$) and post-synaptic ($Y_{l,q}$) traces for each input axon and compartment, respectively; they may all use different exponential filter parameters. These traces enter the learning engine

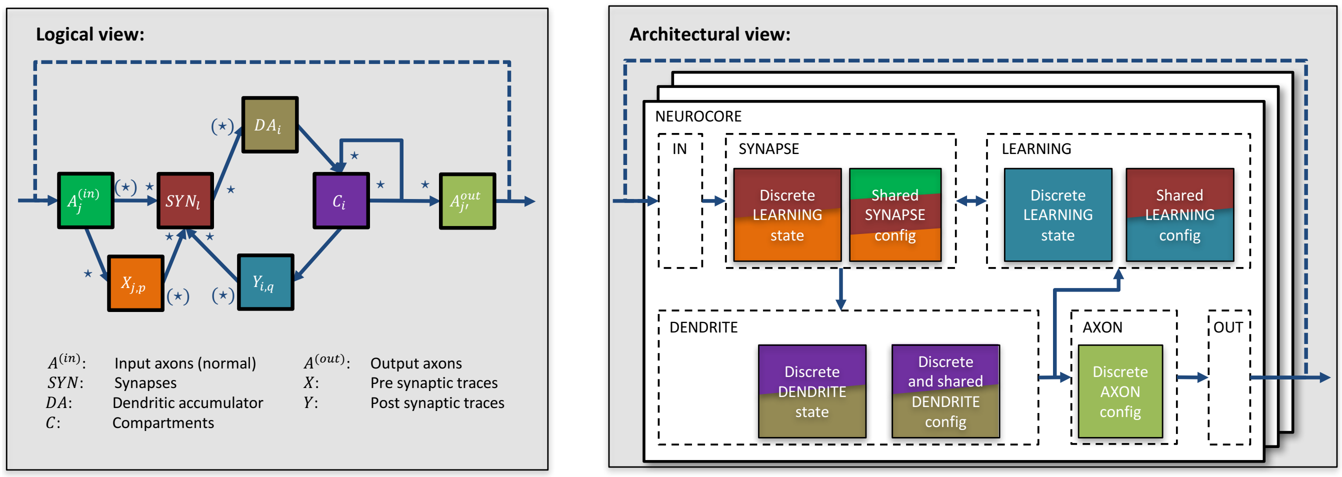

Figure 2. Computational graph underlying Loihi. Logical view (left panel): Logical entities of the system. Each box represents an entity-set such as synapses SYN_j or compartments C_l, while arrows represent their associations. A $\rightarrow^{*}/^{*}\rightarrow$ annotation represents a 1-to-many/many-to-1 association between the entities at both ends of an arrow. When connection sharing is enabled, $(*)$-relations hold as well, i.e. multiple axons reference the same set of synapses. Architectural view (right panel): Dashed boxes within a neuro core represent independent asynchronous circuit blocks. Boxes within these circuit blocks reflect the various state and configuration registers. The color coding indicates how logical entities are decomposed and regrouped within architectural implementation. The IN/OUT blocks are interfaces to the NoC and have no correspondence in the logical view.

as proxies of elapsed time between spikes for spike timing dependent plasticity (STDP) or to correlate pre- to post-synaptic activity.

- **Synapses:** Synapses (SYN_j) embody the connectivity in the neural network. They reference the target compartment's dendritic accumulator (DA_l) and support a weight, as well as optional delay and tag states. Synaptic delays enable advanced temporal codes by delaying the accumulation of an incoming spike while tags are useful as an additional scratch variable within the learning engine. All synaptic states can be modified by the learning engine as a function of synaptic states and pre-/post-traces.
- **Dendritic accumulator:** Input spikes trigger an accumulation of a synaptic weight by a compartment which we refer to as a synaptic activation. The dendritic accumulator (DA_l) acts as a temporal buffer for activations scheduled to be processed at a future algorithmic time step based on the synaptic delay. It also releases the current total activation to the corresponding compartment (C_l).

The architectural implementation in Figure 2 (right panel) partitions the logical entities across neuro cores since each core has limited resources and can hold only a finite number of logical entities. In addition, logical entities are typically decomposed into multiple registers and separated into different independent asynchronous circuit blocks (dashed boxes in Figure 2, right panel) based on circuit optimization considerations. Dynamic state is stored per logical entity. In contrast, static configuration parameters are not stored individually in most cases. Instead, they are stored in shared profile registers per core and referenced by individual logical entities for resource and algorithmic efficiency. The color-coding in Figure 2 reflects how logical entities on the left panel are realized through the various discrete and shared registers on the right panel.

During execution of a SNN, the system of cores operates in three sequential phases within each algorithmic time step:

1. **Neuronal updating and spike processing:** Neuronal and synaptic state is updated locally and in parallel within each neuro core, without the need to access off-chip memory and with neurons only communicating via spikes messages transmitted through the NoC.
2. **Learning:** Synaptic states are updated using a neuro core's integrated learning engine.The learning engine executes microcode that encodes programmable rules specified as a sum of product terms as in Equation (1).
3. **System management (optional):** At the end of each algorithmic time step, there is the option to execute management commands, e.g., to reconfigure or inspect state within neuro cores. This is useful for monitoring and debugging the behavior of a SNN.

4 The Loihi Toolchain

4.1 API

Loihi's Python API is an implementation of the programming model we introduced in Section 2, and allows the programmer to implement SNNs on Loihi in an intuitive way, without requiring intimate knowledge of its architectural details. The API provides ways to (a) define a graph of neurons and synapses and configure their parameters (e.g., decay time constants, spike impulse values, synaptic weights, refractory delays, spiking thresholds, etc); (b) inject external stimulus spikes into the network; (c) implement custom learning rules; and (d) monitor and modify the network state during runtime.

Figure 3 gives an overview of the API class hierarchy. Readers already acquainted with existing SNN simulators such as Brian [10] or libraries such as PyNN [6] will find our API to be comfortably familiar. This is a purposeful design choice to lower the barrier of entry to using Loihi.

4.1.1 Specifying a SNN

Loihi's API abstracts the elements comprising an arbitrary SNN topology into the following Python classes:

- **Net.** The root class for specifying a SNN in the API and the top-level container for all elements of the SNN being configured, i.e., it contains instances of all the other classes described below. Each Loihi API program must have one instance of `Net`. For convenience, it provides several methods for creating and fully-connecting layers of neurons, initialized with reasonable default parameter settings.
- **Compartment.** A `Compartment` models a dendritic compartment of a `Neuron` and is the basic computing element of the SNN.
- **Neuron.** A `Neuron` contains one or more `Compartments`, arranged in a binary dendritic tree. The compartments of a `Neuron` are each configured by calling `configureCompartment(i)` on the i-th `Compartment`. This includes setting such `Compartment` parameters as decay constants, refractory delays, and spiking threshold values, which are components to our variant of the CUBA neuron model.
- **NeuronGroup.** A container class for logically grouping `Neurons`. This is often useful for organizing the SNN topology into neuronal layers.
- **StimulusNode.** A `StimulusNode` models a logical point source that injects spikes into the network.
- **StimulusNodeGroup.** Similar to a `NeuronGroup`, a `StimulusNodeGroup` is a container for logically grouping `StimulusNodes`, e.g., to represent an input layer that injects stimulus from the external world (e.g., a sensor or an image-to-spike encoder) into the `Net`.
- **Synapse.** Each `Synapse` logically belongs to a `Compartment` and is the end point of a `Connection`. When on-chip learning is enabled for a `Synapse`, its states are adjusted according to its assigned learning rule (see `AbstractLearningRule` in Section 4.1.3).
- **SynapseGroup.** A container class for logically grouping `Synapses` for different functional purposes.
- **Connection.** A `Connection` explicitly models a directed edge in the SNN graph where the source vertex is a `Compartment` and the destination vertex is a `Synapse`.

4.1.2 Injecting External Stimulus

To control external spike injection into the SNN, the API provides the following:

- **AbstractExternalStimulus.** An abstract base class that models an external stimulus signal source that injects spikes over designated `Connections` at specific time steps. The intented use case is for programmers to subclass `AbstractExternalStimulus` for their specific application, e.g., to interface with custom signal-to-spike encoders or with physical sensors such as a dynamic vision sensor (DVS) camera [14].

4.1.3 Incorporating Learning

The programmer may implement a wide variety of learning rules through:

- **AbstractLearningRule.** An abstract base class that models an on-chip learning rule, which is expressed in sum-of-products form, as we mentioned above. The programmer subclasses this for her own custom learning rules.
- **RuleGroup.** A container class for logically grouping subclasses of `AbstractLearningRule`. Some networks, e.g., our example in Section 5, operate by dynamically swapping learning rules during runtime; we group these rules into a `RuleGroup`.

4.1.4 Runtime Monitoring and Control

Loihi's API also provides the following facilities for the programmer to record the state of a SNN and control its runtime behavior:

- **Monitor.** A `Monitor` records the state evolution of a SNN during execution. The programmer specifies which network entities' (`Compartments`, `Synapses`, etc) state(s) will be recorded and at what interval. After execution, the programmer may inspect the recorded values for data analysis. This is a useful tool for debugging and validating the behavior of a particular SNN implementation.
- **RuntimeController.** The programmer may require runtime control of the SNN while it is executing. `RuntimeController` allows the programmer to issue commands to modify the state of the SNN at a specific time step.

4.2 Compiler and Runtime Library

Loihi's compiler takes a SNN implementation and produces a common binary bytestream for various backend targets,

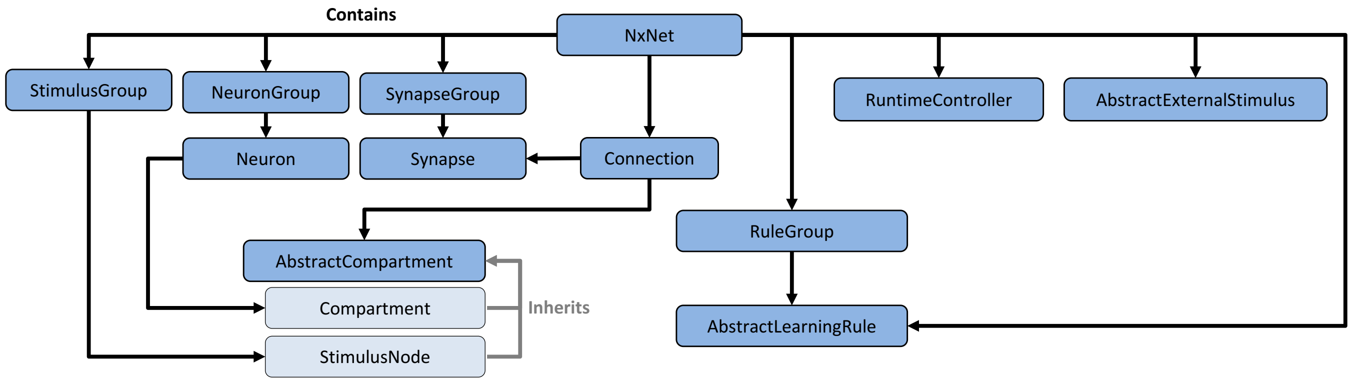

Figure 3. Loihi API class hierarchy

including our functional simulator, FPGA, and of course the chip itself. Compilation proceeds in three phases: preprocessing, resource allocation, and code generation.

In preprocessing, the compiler (1) validates that the network parameters specified are supported by the hardware; (2) extracts the necessary information for producing the shared configuration register entries required by the architecture; and (3) extracts the parameter values required to generate a ruleâĂŹs microcode from the sum-of-products form of that learning rule.

Next, the compiler performs resource allocation by greedily assigning network entities (compartments, synpases, learning rules) in the SNN graph to available core resources. To do so, the compiler statically partitions the SNN graph onto as few cores as possible. Assuming there are enough core resources to accommodate the specified SNN, the compiler next performs code generation by walking through all the network entities in the SNN and outputting the bytestream that encodes each network entity onto a specific neuro core, as well as the shared configuration entries required by that neuro core. The conversion into the binary bytestream is performed by calling into the Loihi runtime library, which provides a low-level interface to controlling the runtime configuration of all the neuro and IA cores that comprise Loihi.

4.3 Target platforms

The Loihi compiler supports several backend targets: our functional simulator, a FPGA implementation of the Loihi architecture, and of course Loihi silicon (Figure 1). Each of these target platforms play vital roles in the creation of and application development on the Loihi architecture:

- The Loihi functional simulator implements the computational graph in Section 3 in software. It is functionally bit-accurate with respect to the FPGA and silicon implementations of Loihi and offers full visibility into the dynamical system state. Thus, it not only supports and eases hardware validation but also enables rapid algorithmic feature exploration as well as future hardware prototyping.
- The FPGA platform emulates the Loihi silicon and is thus crucially important for validation and hardware feature exploration. Beyond this development aspect, it can also be integrated with the Loihi silicon for rapid generation and deployment of architectural Loihi variants that are optimized for specific workloads, i.e., trade off efficiency of the hardware for programmability.
- The silicon target ultimately implements the Loihi architecture in the most energy-efficient, performance-optimized and memory-dense way possible.

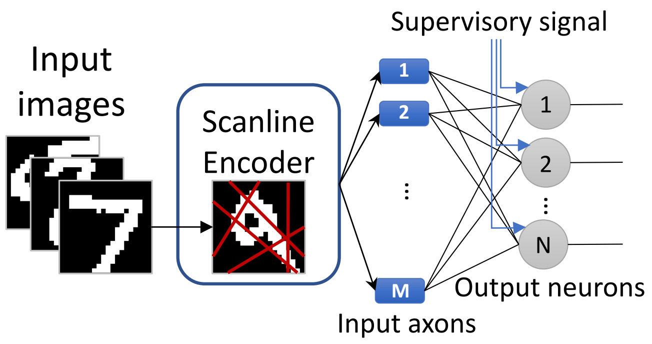

Figure 4. Preprocessing steps and SNN model for our image classifier. The scan line encoder is an image edge detector that outputs spikes.

5 Example: Single-layer SNN for Supervised Image Classification

Having introduced the toolchain, we now give an end-to-end example that uses it to program a SNN onto Loihi. The example SNN we will implement has a single layer of output neurons (Figure 4) and can be trained in a supervised manner to classify images from the MNIST database [9]. We will first describe the model and then the training and inference implementations.

5.1 Model Description

All images for both training and inference are first preprocessed and encoded into spikes before being injected into the SNN. The preprocessing consists of binarizing each image vector and encoding it with a so-called *scan line* encoder, an edge detection procedure inspired by human saccadic eye movements. This encoding exploits the large degree of topological redundancy in natural images by sampling each image sparsely along certain directions ("scan lines", in red in Figure 4), while being robust against small local image deformations such as feature shifts or stretches. The encoder produces spikes along each scan line where the scan line is incident on an edge in the image. We present each encoded training image to a single-layer SNN and train it by applying a supervisory signal (via the learning rules) to the output neuron whose index matches the ground truth label of the input image. During inference, the index of the most active output neuron (as identified by spike count) is the predicted class of the current input.

As a supervised learning rule, we use an event-based formulation of the Widrow-Hoff rule [15], which we call supervised STDP (S-STDP) and write in the sum-of-products form understood by the learning engine (as in Equation (1)):

$$W_{i,j}(t) = W_{i,j}(t-1) + y_{i,0} \cdot S_1 \cdot x_{j,1} + u_\kappa \cdot S_2 \cdot x_{j,1} \quad (2)$$

$$W_{i,j}(t) = W_{i,j}(t-1) + y_{i,0} \cdot S_1 \cdot x_{j,1} \quad (3)$$

Here, t is the time step, $W_{i,j}$ is the synaptic weight between the j-th input (pre-synaptic) and i-th output (post-synaptic) neurons, and $S_1 < 0,\ S_2 > 0$ are constants. The actual output of the network is represented by the post-synaptic (i.e., output) spike sequence $y_{i,0} \in 0, 1$ and the the supervisory signal is encoded into a sequence of periodic training impulses, $u_\kappa \in 0, 1$ with periodicity $\tau^{(epoch)} \cdot 2^\kappa$. The execution of both learning rules is gated on the pre-synaptic trace $x_{j,1} > 0$, i.e. the j-th input neuron being stimulated by the current input spike signal.

Synapses whose post-synaptic neuron corresponds to the class of the current input image use the potentiating rule in Equation (2), while all others use the depressive-only rule in Equation (3). As a result, each neuron develops a selectivity for the difference between the average in-class and the average out-of-class image vectors.

5.2 Training

We implement this SNN using Loihi's API as follows. First, we create a `Net` container for the entire network and instantiate a `StimulusNodeGroup` representing the input layer and a `NeuronGroup` for the output layer in Figure 4.

```
snn = Net()
numExtInputs = 1920
inputLayer = snn.createStimulusNodeGroup(
    numExtInputs)
numNeurons = 10
outputLayer = snn.createNeuronGroup(
    numNeurons,
    threshold=1024,
    decayU=410,
    decayV=410,
    refractory=2)
```

Here the parameters to `createNeuronGroup` correspond to the activation threshold, the two compartment trace decay constants, and the refractory period for each `Neuron` being instantiated in the `NeuronGroup`. These parameters are required by the variant of the CUBA model that Loihi implements.

Next we, instantiate the two learning rules described by Equations (2) and (3), and connect the input with the output layer.

```
p_rule = SSTDPPotentiationRule(
            productTerms=[('Y0', 'X1'), ('X1',)]
            coeffs=[-1, 1],         # S_1, S_2
            coeff_exps=[-5, -5],
            preTraceDecay=[10,],
            preTraceAcc=[127, ])
d_rule = SSTDPDepressionRule(
            productTerms=[('Y0', 'X1')]
            coeffs=[-1],            # S_1
            coeff_exps=[-5],
            preTraceDecay=[10,],
            preTraceAcc=[127, ])
rules = SSTDPRuleGroup(
    ruleList=[d_rule, p_rule])
snn.fullyConnect(inputLayer, outputLayer, rules)
```

As can be seen by the syntax of the initialization parameter `productTerms`, the learning rules directly reflect the sum-of-product terms in Equations (2) and (3), respectively. In both cases, we have also used learning rule parameters for $S_1 = -2^{-5}$ and $S_2 = 2^{-5}$, which are specified as the `coeff` and `coeff_exp` parameters.

For training, we encode all the training set images via `LineScanStimulus`, a subclass of `AbstractExternalStimulus` that implements the scan line encoder described above.

```
lss = LineScanStimulus(training_images,
      inputLayer, snn.connections)
snn.addExternalStimulus(lss,
      lss.getDuration())
```

For training, we will need to dynamically reassign the appropriate learning rule (2) or (3) to the synapses of different output neurons at runtime. To achieve this, we create and initialize a `RuntimeController` with the the `groundTruthLabels` of the training set. This assignment allows the `RuntimeController` to issue the proper network reconfiguration commands for each image during execution. For example, an input image with ground truth label i results in the `RuntimeController` assigning the `p_rule` to the synapses of the i-th output neuron whereas all the other synapses are assigned the `d_rule`. Thanks to indirection in Loihi's learning rule assignments, the rule changes can be applied with just a single register write per neuron.

```
if rtctl is None:
    rtctl = RuntimeController(
        timeoffset=GUARD_INTERVAL,
        timestepping=TIMESTEPS_PER_IMAGE)
    snn.addRuntimeController(rtctl)
rtctl.appendLabels(groundTruthLabels)
```

Finally, because we will want to export the trained model (i.e., the synaptic weights) after the SNN is trained, we set up a `Monitor` on the synapses and set the recording interval `dt` such that only the last time step is recorded (it is, of course, possible to set it to a shorter interval to record intermediate weights during training).

```
mon = Monitor([synapse
               for synapseGroup
               in snn.synapseGroups
               for synapse in synapseGroup],
              [SynapseState.WEIGHT],
              dt=num_instances*TIMESTEPS_PER_IMAGE
                 +GUARD_INTERVAL)
snn.addMonitor(mon)
```

To execute this SNN for training, we first invoke the compiler on the Python program. This produces a bytestream (in our case a binary file) that can be loaded and run on Loihi or, for example, the functional simulator. Below, we launch the simulator and, after the simulation completes, we save the trained weights for future use during inference.

```
sim = PySimulator.Simulator(snn)
sim.simulate(nsteps, reportProgress=True)
wgts = mon.getStateValues(sim, SynapseState.WEIGHT)
numpy.savetxt(wgtsFilename,
    wgts.values()[0], fmt="%d")
```

5.3 Inference

For inference, we'd instantiate the same SNN toplogy as above, but instead load the previously trained weights:

```
snn.fullyConnect(inputLayer, outputLayer,
    weight=numpy.loadtxt(wgtsFilename,
    dtype=int).reshape(numExtInputs,numNeurons))

neuronIdx = [neuron.compartments[0]
    for neuron in outputLayer]
spikemon = Monitor(neuronIdx,
    [CompartmentState.SPIKES])
snn.addMonitor(spikemon)
```

Above, we have also added a spike `Monitor` to record output spikes in order to determine the activity of the neurons in the output layer and thus the classification result.

During the inference phase, we encode all test images using `LineScanStimulus`. After each image's spike train is injected into the SNN, we reset compartment state; this is scheduled by providing the appropriate reset time steps to the `RuntimeController.clearAllCx()` method.

```
lss = LineScanStimulus(test_images,
    inputLayer, snn.connections)
snn.addExternalStimulus(lss,
    lss.getDuration())

rtctl = RuntimeController(
    timeoffset=GUARD_INTERVAL,
    timestepping=TIMESTEPS_PER_IMAGE)
rtctl.clearAllCx(range(GUARD_INTERVAL,
    len(imglist)*TIMESTEPS_PER_IMAGE,
    TIMESTEPS_PER_IMAGE))
snn.addRuntimeController(rtctl)

sim = PySim.Simulator(snn)
numSteps = GUARD_INTERVAL+
    len(imglist)*TIMESTEPS_PER_IMAGE
sim.simulate(numSteps)

out = spikemon.getStateValues(sim,
    CompartmentState.SPIKES)
```

The inference results are shown in Figure 5. The top panel shows the scan line spike encoding corresponding to each input image as it is injected into the network (blue dots indicate spikes). The bottom panel shows the SNN's output spikes in red, overlaid with the corresponding classification result. The trained network correctly classifies most inputs, achieving a classification accuracy of 96.4% on the standard MNIST test set.

6 Conclusions

We have presented an end-to-end toolchain for programming Loihi, our novel neuromorphic processor. The toolchain includes a Python-based API that allows the programmer to quickly implement SNNs capable of both learning and inference; a compiler and runtime for building and executing these programs on our platform; and several target Loihi platforms. We envision that this toolchain will not only lower

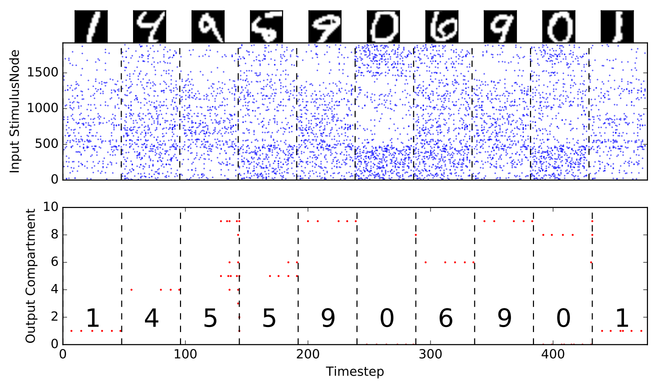

Figure 5. Inference results for the single-layer SNN previously trained on the MNIST data set. The top panel shows the input image and the corresponding spike encoding (blue). The bottom panel shows the output spiking activity (red) with the classification result overlaid. In this example, the SNN correctly classifies all but the third image.

the barrier for the the research community to develop neuromorphic applications on the current generation of Loihi, but also serve as a means to collaboratively explore new features for future hardware generations.

References

[1] Johannes Schemmel, Daniel Briiderle, Andreas Griibl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In IEEE ISCAS, pages 1947–1950. IEEE, 2010.

[2] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. Proceedings of the IEEE, 102(5):699–716, 2014.

[3] Steve Furber, David Lester, Luis Plana, Jim Garside, Eustace Painkras, Steve Temple, and Andrew Brown. Overview of the spinnaker system architecture. IEEE Transactions on Computers, 62(12):2454–2467, 2013.

[4] Paul Merolla, John Arthur, Rodrigo Alvarez-Icaza, Andrew Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. Science, 345(6197):668–673, 2014.

[5] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steve McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: a neuromorphic manycore processor with on-chip learning. IEEE Micro Magazine, submitted.

[6] Andrew P Davison, Daniel Brüderle, Jochen M Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. Frontiers in Neuroinformatics, 2:11, 2009.

[7] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. Nengo: a python tool for building large-scale functional brain models. Frontiers in Neuroinformatics, 7:48, 2014.

[8] Arnon Amir, Pallab Datta, William Risk, Andrew Cassidy, Jeffrey Kusnitz, Steve Esser, Alexander Andreopoulos, Theodore Wong, Myron Flickner, Rodrigo Alvarez-Icaza, et al. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In IEEE IJCNN, pages 1–10. IEEE, 2013.

[9] Yann LeCun, Corrina Cortes, and Christopher Burges. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[10] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in python. Frontiers in Neuroinformatics, 2, 2008.

[11] Michael Beyeler, Kristofor Carlson, Ting-Shuo Chou, Nikil Dutt, and Jeffrey Krichmar. Carlsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. In IEEE IJCNN, pages 1–8. IEEE, 2015.

[12] Bartlett Mel. Synaptic integration in an excitable dendritic tree. Journal of Neurophysiology, 70(3):1086–1101, 1993.

[13] Tim Vogels and Larry Abbott. Signal propagation and logic gating in networks of integrate-and-fire neurons. Journal of Neuroscience, 25(46):10786–10795, 2005.

[14] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128×128 120 db 15μs latency asynchronous temporal contrast vision sensor. IEEE Journal of Solid-State Circuits, 43(2):566–576, 2008.

[15] Filip Ponulak and Andrzej Kasiński. Supervised learning in spiking neural networks with resume: sequence learning, classification, and spike shifting. Neural Computation, 22(2):467–510, 2010.