SNN Simulator Worksheet and Research Notes

Lucas S Hindman Kurtis D Cantley

This document provides a capability summary worksheet for the open-source spiking neural network (SNN) simulators surveyed in *Finding Order in CHAOSS*, followed by detailed research notes for each simulator. Each entry in the worksheet links to its corresponding notes section via the **[notes]** reference.

Table 1: Open-Source SNN Simulator Worksheet

Simulator	Bio-Inspired	Bio-Plausible	Integrate-and-Fire	Compartmental	Equation-Based	Supervised	Unsupervised	Static	CPU	GPU	FPGA	Neuromorphic	Freshness	Documentation	Community	Viability Score
ANNarchy [notes]	✓	✓	✓		✓		✓		✓	✓			2	2	2	6
Arbor [notes]			✓				✓		✓	✓			2	2	2	6
Auryn [notes]			✓				✓		✓				2	2	1	5
BindsNET [notes]	✓		✓				✓	✓	✓	✓			2	2	2	6
BrainCog [notes]	✓	✓	✓	✓		✓	✓		✓	✓	✓		2	2	2	6
BrainPy [notes]		✓	✓			✓			✓	✓			2	2	2	6
Brian2 [notes]				✓	✓		✓		✓	✓			2	2	2	6
Brian2Loihi [notes]			✓				✓		✓			✓	1	1	1	3
BSim [notes]			✓					✓	✓	✓			0	0	1	1
CARLsim [notes]	✓		✓			✓	✓		✓	✓			2	2	2	6
cuSNN [notes]			✓				✓	✓		✓			0	1	1	2
Doryta [notes]			✓					✓	✓				1	1	0	2
EDLUT [notes]	✓	✓	✓				✓	✓	✓	✓			2	0	1	3
Emergent [notes]									✓	✓			2	2	2	6
ENLARGE [notes]			✓					✓		✓			1	1	1	3
Fugu [notes]			✓					✓	✓			✓	2	2	1	5
Genesis [notes]		✓		✓			✓		✓				0	2	1	3
GeNN [notes]	✓	✓			✓		✓		✓	✓			2	2	2	6
jaxSNN [notes]			✓					✓	✓			✓	2	2	1	5
Lava [notes]			✓				✓		✓			✓	2	2	2	6
Memristor-Spikelearn [notes]			✓				✓		✓				2	1	0	3
N2S3 [notes]			✓				✓		✓			✓	0	2	2	4
Nengo [notes]	✓		✓				✓		✓	✓	✓	✓	2	2	2	6
NengoDL [notes]	✓		✓			✓		✓	✓	✓			2	2	2	6
NEST [notes]	✓	✓	✓	✓	✓		✓	✓	✓				2	2	2	6
NEST GPU [notes]		✓	✓		✓		✓			✓			2	2	1	5
Neuron [notes]		✓	✓	✓			✓		✓				2	2	2	6
NeuroPack [notes]	✓		✓			✓	✓		✓				1	1	0	2
NEUTRAMS [notes]			✓					✓	✓			✓	0	0	0	0
Norse [notes]	✓		✓				✓		✓	✓		✓	2	2	2	6
PymoNNto [notes]	✓	✓	✓			✓	✓		✓	✓			2	2	1	5
PymoNNtorch [notes]	✓	✓	✓			✓	✓		✓	✓			2	2	1	5
PySNN [notes]			✓				✓		✓	✓			0	2	1	3
RANC [notes]			✓			✓			✓		✓		1	0	1	2
RAVSim [notes]			✓						✓				2	2	0	4
SaRNN [notes]			✓					✓	✓			✓	2	0	0	2
SINABS [notes]			✓			✓		✓	✓			✓	2	2	2	6
SNNToolbox [notes]			✓					✓	✓				1	2	2	5
snnTorch [notes]			✓			✓			✓	✓		✓	2	2	2	6
SPAIC [notes]	✓	✓	✓			✓	✓		✓	✓			2	2	1	5
SPIKE [notes]			✓			✓		✓		✓			0	2	1	3
SpikingJelly [notes]			✓			✓	✓	✓	✓	✓		✓	2	2	2	6
SpykeTorch [notes]			✓				✓		✓	✓			1	2	0	3
Spyx [notes]			✓			✓				✓		✓	2	2	1	5
SuperNeuro [notes]			✓				✓		✓	✓			2	1	1	4
	Neuron Model Family					Learning Method			Hardware Support				Viability Scorecard			

Simulator Research Notes

ANNarchy Research Notes

ANNarchy is a neural simulator designed for simulating networks of spiking neurons on parallel hardware, supporting both rate-coded and spiking networks, or combinations of both. It uses an equation-oriented mathematical approach (similar to the Brian simulator) and generates optimized C++ code for efficient simulations on multi-core systems or GPUs. The Python interface is modeled after PyNN, allowing users to easily define neuron and synapse models. ANNarchy supports various standardized neuron models, including integrate-and-fire neurons with exponential and alpha-shaped conductances, adaptive integrate-and-fire neurons like the Izhikevich model, and Hodgkin-Huxley neurons. Synapse models include short-term plasticity (STP) and spike-timing dependent plasticity (STDP), along with rate-coded models such as leaky-integrator neurons and Hebb, Oja, and IBCM synapses.

Though ANNarchy was originally published in 2015 and is outside the last eight-year inclusion window, it remains highly relevant due to its citation record and ongoing development, with version 4.8.1 released in July 2024. It has accumulated over 115 citations, with recent research using ANNarchy as a baseline for new simulator advancements, such as in the PymoNNto and PymoNNtorch projects. This active development and consistent application in current research justify its inclusion in the systematic review as a valuable, flexible tool for constructing complex neural networks.

ANNarchy at a Glance

Neuron Models: Izhikevich, Leaky Integrate-and-Fire, Hodgkin–Huxley, Equation-based, Rate-coded Leaky Integrator
Learning Methods: Spike Timing Dependant Plasticity (STDP), Short-Term Plasticity (STP), Equation-based, Rate-coded - Oja, Rate-coded - IBCM, Rate-coded - Hebb
Targeted Hardware: CPU, GPU
Source Code: github.com/ANNarchy/ANNarchy

Arbor Research Notes

Arbor is a library designed for performance-portable network simulations of multi-compartment neuron models, supporting both the NMODL and NeuroML description languages. Funded as part of the Human Brain Project, Arbor is under active development and focuses on computational neuroscience simulations. It is specifically tailored for modeling morphologically-detailed cells, from single models to large-scale networks, and offers a wide range of parameters to define the behaviors of both neurons and synapses. While machine learning does not appear to be a central focus of Arbor, it provides extensive documentation and tutorials, including resources for implementing spike-timing dependent plasticity (STDP).

The library is designed to facilitate complex, detailed simulations of biological neural networks, with an emphasis on scalability and precision in representing the structure of neurons. Arbor's ability to handle multi-compartment cells and large networks makes it suitable for research in computational neuroscience. Its active development and broad support for neuron and synapse parameterization ensure that users can model a wide variety of neural behaviors, contributing to the understanding of brain-like systems and networks.

Arbor at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, GPU, GPU Cluster
Source Code: github.com/arbor-sim/arbor

Auryn Research Notes

The Auryn simulator, first published in 2014, remains relevant due to its continued active maintenance, with version 0.8.3rc released in 2023. Despite falling outside the eight-year inclusion window for this review, it has garnered over 95 citations, including its use in a 2023 Nature Neuroscience study by Halvagal and Zenke. Auryn is included in this systematic review due to its ongoing relevance in spiking neural network (SNN) research. The simulator offers benchmark comparisons with other popular tools such as Neuron, Brian, and NEST, revealing key insights into performance limitations, particularly with communication latency and the specifics of implementing spike-timing-dependent plasticity (STDP).

Auryn excels in simulating plastic synapses in recurrent neural networks, with a focus on STDP in an event-based manner for weight updates during pre- or postsynaptic spikes. While current off-the-shelf hardware can support real-time simulations of synaptic plasticity, Auryn's performance can be limited by communication latency between nodes and memory. Specialized hardware is needed for simulations that exceed real-time speeds, enabling more complex studies of synaptic plasticity and long-term learning processes. The simulator offers a flexible and optimized environment for these types of studies, particularly for those focused on high temporal resolution and long-term plasticity.

Auryn at a Glance

Neuron Models: Adaptive Exponential Integrate-and-Fire (aEIF), Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Short-Term Plasticity (STP), Triplet STDP
Targeted Hardware: CPU
Source Code: github.com/fzenke/auryn

BindsNET Research Notes

BindsNET is a Python library designed for simulating spiking neural networks (SNNs), with a focus on machine learning and reinforcement learning applications. Built on PyTorch for efficient matrix computations, it enables rapid prototyping and provides a user-friendly syntax. The library supports various hardware platforms and includes a Network object to coordinate Nodes and Connections, offering flexibility in developing complex network structures. It also provides modules for datasets, environment interactions, and evaluation methods for unsupervised learning, with future plans to expand to additional read-out methods.

While BindsNET simplifies the process by offering a selection of popular neuron models, it does not support specifying neural dynamics through arbitrary differential equations. It supports various learning rules such as Hebbian learning and spike-timing-dependent plasticity (STDP), providing tools for exploring reinforcement learning and machine learning tasks using SNNs. The library prioritizes high-level functionality, offering flexibility to control individual neural components, making it suitable for a range of real-world applications.

BindsNET at a Glance

Neuron Models: Izhikevich, Integrate-and-Fire, Leaky Integrate-and-Fire, McCulloch-Pitts
Learning Methods: Spike Timing Dependant Plasticity (STDP), Reward-Modulated (M-STDP) (R-STDP), Reward-Modulated Elegibility Trace (M-STDP-ET) (R-STDP), Basic Hebbian, ANN-to-SNN Conversion
Targeted Hardware: CPU, GPU
Source Code: github.com/BindsNET/bindsnet

BrainCog Research Notes

BrainCog is an integrated platform designed to bridge the gap between brain simulations and brain-inspired AI, addressing the distinct goals of each field. Built on PyTorch, BrainCog leverages CPU and GPU capabilities to simulate brain functions and develop brain-inspired AI, providing foundational components and supporting diverse cognitive functions. Its architecture supports various models for spiking neural networks (SNNs), enabling the creation of brain-inspired SNN models with flexible neuron types and learning rules. BrainCog also incorporates an FPGA accelerator, Firefly, designed to process a wide range of BrainCog models with fewer hardware constraints than typical neuromorphic systems, which often limit neuron models, encoding, and learning rules.

Aiming to promote coordination between brain simulation and AI, BrainCog enables multi-scale simulations, from neural microcircuits and cortical columns to whole-brain structures, with models covering species like mice and macaques. The platform offers biologically plausible learning rules and plasticity models, such as Hebbian learning and short-term plasticity, supporting complex cognitive functions like multisensory integration and music composition. These capabilities demonstrate BrainCog's potential as a tool for accurate, efficient simulations and as a resource for exploring applications of brain-inspired AI in areas like music learning and creation.

BrainCog at a Glance

Neuron Models: Izhikevich, Integrate-and-Fire, Leaky Integrate-and-Fire, Hodgkin–Huxley, Adaptive Exponential Integrate-and-Fire (aEIF), Multi-compartment
Learning Methods: Spike Timing Dependant Plasticity (STDP), Basic Hebbian, Short-Term Plasticity (STP), Surrogate Gradient Back Propogation, Reward-Modulated Elegibility Trace (M-STDP-ET) (R-STDP)
Targeted Hardware: CPU, GPU, FPGA
Source Code: github.com/BrainCog-X/Brain-Cog

BrainPy Research Notes

BrainPy is a flexible framework designed for brain dynamics programming, providing an integrated platform for building, simulating, training, and analyzing models of brain activity with high performance across CPU, GPU, and TPU devices. Leveraging JIT compilation via JAX, BrainPy enables efficient model execution, though the limitations of this approach on model types are not clearly addressed. The framework supports modular and composable programming, allowing users to define brain dynamics models at various levels and combine them hierarchically, making it a powerful tool for multi-scale brain modeling and simulation.

Despite the documentation providing examples and tutorials for model building, it may be challenging for users outside neuroscience, as the framework appears more focused on brain simulation than brain-inspired AI. BrainPy offers essential components such as mathematical operators, differential equation solvers, and object-oriented JIT compilation, giving users the flexibility to freely

define brain dynamics models and achieve high performance without modifying their code. This structure positions BrainPy as a valuable resource for neuroscience applications, emphasizing brain simulation over broader AI applications.

BrainPy at a Glance

Neuron Models: Leaky Integrate-and-Fire, Hodgkin–Huxley
Learning Methods: Back Propogation Through Time (BPTT), Ridge Regression, FORCE Training (Reservoir Computing)
Targeted Hardware: CPU, GPU, TPU
Source Code: github.com/brainpy/BrainPy

Brian2 Research Notes

Brian2 is a neural simulator that uses differential equations to describe neuron behavior and learning rules, similar to frameworks like ANNarchy, NEST, and NEURON, providing flexibility in neural network construction. Written in C++ with a Python wrapper, Brian2 relies on runtime code generation to convert high-level model descriptions into efficient low-level code. This approach enables performance optimization, allowing models to run in either "runtime" mode (Python-controlled) or "standalone" mode (independent of Python) for greater efficiency. With the Brian2GeNN tool, models can also be executed on NVIDIA GPUs, as it translates Brian2 spiking neural networks (SNNs) into GeNN-compatible code.

While Brian2's code generation boosts performance, modifying neuron behavior may be challenging if the desired dynamics aren't readily available. However, it offers a way to load custom C++ libraries and write C++ functions directly in Python, enhancing customization options. Brian2 supports complex applications, including real-time audio processing from live microphone input, showcasing its capacity for real-world, time-sensitive tasks. Overall, Brian2 provides a flexible and efficient platform for developing reproducible computational experiments, addressing both performance and flexibility challenges in neural simulation.

Brian2 at a Glance

Neuron Models: Equation-based, Multi-compartment
Learning Methods: Equation-based, Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, GPU
Source Code: briansimulator.org/

Brian2Loihi Research Notes

Brian2Loihi is an open-source emulator for the Loihi neuromorphic chip, built upon the spiking neural network simulator Brian. It provides a user-friendly prototyping tool for developing intelligent neuromorphic solutions by emulating Loihi's computational unit. While the emulator accurately models Loihi's behavior for small models and offers a rapid startup time, it does not include all Loihi features, such as the homeostasis mechanism, rewards, and tags for learning rules. Detailed mathematical models of Loihi's hardware are provided, which is a significant benefit for understanding and developing on Loihi's platform.

For smaller models, Brian2Loihi is competitive in performance, making it an excellent tool for prototyping. However, for larger and longer-running models where runtime becomes a limiting factor, the actual Loihi hardware outperforms the emulator. The paper discusses the emulator's ability to accurately simulate Loihi's computational unit, including on-chip learning with minimal

error due to stochastic rounding, although it highlights that certain discrepancies may arise in more complex simulations.

Brian2Loihi at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Equation-based, Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, Loihi
Source Code: github.com/sagacitysite/brian2_loihi

BSim Research Notes

BSim is a high-performance simulation framework for Spiking Neural Networks (SNNs) that focuses on leveraging GPGPUs to accelerate simulation. It utilizes code generation to produce C++ and CUDA code, and the framework includes precompiled libraries that implement standard models in PyNN, though its benchmarking was primarily done using the Leaky Integrate-and-Fire (LIF) neuron model. The paper does not specify any particular learning mechanisms, and the source code only contains implementations for static synapses. The optimization methods used in BSim, such as fine-grained network representation, cross-population-projection parallelism, and sparsity-aware load balancing, aim to improve parallelism, memory access patterns, and load balance for better performance.

However, one limitation in the benchmarking results is that BSim optimized its testing for the LIF neuron model, which GeNN does not natively support. To compare performance against GeNN, they used a toolchain that involved converting a LIF neuron model defined in Brian2 into a form compatible with GeNN via Brian2GeNN. This approach means the benchmark results are not solely comparing BSim against GeNN, but also factoring in the additional complexity of the conversion process, potentially impacting the accuracy of the comparison.

BSim at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Static Synapse (No Learning)
Targeted Hardware: CPU, GPU, GPU Cluster
Source Code: github.com/CRAFT-THU/BSim

CARLsim Research Notes

CARLsim 6 is a significant update to the Spiking Neural Network (SNN) simulation platform, designed for biologically plausible modeling at scale. It supports advanced features such as configurable spike-timing dependent plasticity (STDP), short-term plasticity (STP), and multiple neuromodulators that influence neural excitability and synaptic plasticity, enhancing learning. These neuromodulators play a crucial role in long-term potentiation (LTP) and long-term depression (LTD), allowing STDP-based learning to adapt to environmental needs. The platform now includes an automated parameter tuning interface (PTI) integrated with evolutionary computation tools, Docker support, and Python-based analysis tools, making it more powerful for neuromorphic computing and cognitive machine development.

While CARLsim does not natively support neuromorphic hardware, it integrates cycle-accurate models of hardware like TrueNorth, Loihi, and DynapSE into PyCARL for hardware-software co-simulation. This allows for the simulation of large-scale networks and the exploration of hardware-software interactions. Additionally, CARLsim 6 supports multi-core computers and GPUs, significantly improving its ability to simulate complex networks efficiently. The introduction of support

for LIF neurons in CARLsim 5 and the continued development of its biologically realistic modeling capabilities make CARLsim a valuable tool for advancing the field of neuromorphic computing.

CARLsim at a Glance

Neuron Models: Izhikevich, Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Glutamatergic Synapses (E-STDP), GABAergic Synapses (I-STDP), Dopamine-modulated STDP (DA-STDP), Evolutionary Parameter Tuning, Short-Term Plasticity (STP)
Targeted Hardware: CPU, GPU
Source Code: github.com/UCI-CARL/CARLsim6

cuSNN Research Notes

The cuSNN simulator was developed specifically for the optical flow estimation work presented in the paper, which focuses on hierarchical spiking neural networks (SNNs) for motion perception using event-based vision sensors. The network incorporates an adaptive neuron model and stable spike-timing-dependent plasticity (STDP) for unsupervised learning, enabling feature extraction and motion perception. It uses convolutional and fully-connected layers to process different functions, with a modified LIF neuron model and a novel STDP implementation. Although the authors do not elaborate on why they created a custom simulator, it is likely that the limited availability of GPU-accelerated SNN simulators at the time (2018) influenced this decision.

While the cuSNN simulator's simplicity may limit its broader applicability, it provides valuable insights into GPU-based simulation techniques for SNNs. The research highlights the potential for hierarchical architectures in motion perception tasks, demonstrating how SNNs can efficiently process visual information through feature extraction and motion selectivity. However, the specific utility of the cuSNN simulator for other projects, such as those in the ENDS lab, remains uncertain due to its focused design for optical flow estimation.

cuSNN at a Glance

Neuron Models: Leaky Integrate-and-Fire, Adaptive Integrate-and-Fire
Learning Methods: Static Synapse (No Learning), Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: GPU
Source Code: github.com/tudelft/cuSNN

Doryta Research Notes

Doryta is a simple C-based simulator for spiking neural networks (SNNs) focused on LIF neurons, primarily designed to replicate typical neuromorphic hardware behavior rather than biological neuron functions. It does not include learning mechanisms and serves more as a runtime environment for SNN models. The authors implemented two neural network applications to evaluate Doryta's capabilities: Conway's Game of Life, demonstrating Turing completeness, and MNIST classification using the LeNet model. This work may be relevant for exploring general-purpose computing with SNNs, especially in applications like spintronic hardware, as it evaluates the energy and latency benefits of spintronic-based SNNs using parallel discrete-event simulation (PDES).

The Doryta simulator uses the ROSS parallel simulation engine and supports models trained in Whetstone, although creating custom translation procedures for Doryta can be costly due to the lack of standardized frontends. The paper also emphasizes the importance of optimizing interconnects between neurons in spintronic chips, as energy consumption is largely driven by interconnects rather

than the neurons themselves. While Doryta's primary focus is on spintronic hardware, its potential for future integration with other frameworks like PyNN could enhance its accessibility and broaden its use in neuromorphic computing research.

Doryta at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Static Synapse (No Learning)
Targeted Hardware: CPU
Source Code: github.com/helq/doryta

EDLUT Research Notes

EDLUT is a spiking neural network simulator initially released in 2006, but included in this review due to its over 60 citations and ongoing development, with the most recent version, 3.0, released in 2023. EDLUT's novel hybrid approach leverages both CPU and GPU resources for simulations, allowing it to support both event-driven and time-driven computation. Event-driven simulations are optimized for simple, low-activity models like leaky integrate-and-fire (LIF), while time-driven methods are suited to more complex, high-activity models. Although EDLUT supports spike-timing-dependent plasticity (STDP), the primary study referenced uses static synaptic weights, read from configuration files.

Despite its unique approach, EDLUT suffers from limited documentation, with no tutorials or comprehensive guides, relying on the project paper and GitHub resources for user information. The simulator's hybrid design enables the efficient simulation of large-scale networks, with low-activity areas processed on the CPU and high-activity subsystems allocated to the GPU. Users define spike encoding through an INPUT.DAT file, specifying the spike train's start time, spike count, and interval between spikes. Overall, EDLUT's architecture demonstrates a modular approach for simulating layered neural networks, balancing efficiency across different simulation methods to support diverse neural models.

EDLUT at a Glance

Neuron Models: Leaky Integrate-and-Fire, Hodgkin–Huxley, Izhikevich
Learning Methods: Spike Timing Dependant Plasticity (STDP), Static Synapse (No Learning)
Targeted Hardware: GPU, CPU
Source Code: github.com/EduardoRosLab/edlut

Emergent Research Notes

Emergent is a neural network simulator implemented in the Go programming language, using the Go shader language (gosl) to support GPU processing. This approach allows Emergent to avoid the common need for a separate wrapper language (such as in Python/C++ frameworks), as Go is sufficiently fast and efficient for direct simulation. The use of gosl ensures a unified codebase that can handle both CPU and GPU implementations without the need for distinct CUDA code, simplifying development and maintenance.

Designed specifically to complement the Computational Cognitive Neuroscience textbook, Emergent provides example models and exercises geared toward teaching the Leabra neural network model, a focus that distinguishes it from other simulators. This educational orientation makes Emergent a unique tool, particularly suited for those learning computational neuroscience through hands-on experimentation with Leabra-based models.

Emergent at a Glance

Neuron Models: Leabra
Learning Methods: Leabra
Targeted Hardware: CPU, GPU
Source Code: github.com/emer/emergent

ENLARGE Research Notes

ENLARGE is an open-source spiking neural network (SNN) simulation framework optimized for GPU clusters with multiple GPUs per compute node. It focuses on the efficient simulation of point neuron models, such as LIF neurons with static synapses, and is specifically designed to address the challenges of large-scale networks by optimizing computation, communication, and synchronization across different levels (inside-GPU, intra-node, and inter-node). Although the framework does not include built-in learning mechanisms, it supports user-defined models and rate-based encoding, as demonstrated in its performance benchmarks.

The framework operates primarily on CUDA GPUs, with CPUs used only for communication tasks. ENLARGE's multi-level architecture minimizes communication and synchronization costs, enabling faster simulations of large-scale SNNs. By optimizing the computational aspects of network simulations and targeting GPU clusters, ENLARGE enhances performance for large-scale SNN simulations, making it suitable for researchers working with complex, large-scale neuromorphic models.

ENLARGE at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Static Synapse (No Learning)
Targeted Hardware: GPU, GPU Cluster
Source Code: github.com/yhx/enlarge

Fugu Research Notes

Fugu is a framework designed to simplify the programming, configuration, and deployment of neuromorphic systems, offering a hardware-agnostic approach through intermediate representations (IM) of spiking neural networks (SNNs). It generates a NetworkX graph representation, which is intended to be easily compiled for various hardware platforms as they become more standardized. Fugu provides a basic LIF neuron model but does not include learning mechanisms, though it offers a reference implementation of an SNN software simulator for algorithm development. The framework's primary focus is to enable users to work with neuromorphic systems without needing deep knowledge of specific hardware architectures.

The framework supports the development of scalable spiking neural algorithms and provides a flexible method for defining computational graphs known as scaffolds. These scaffolds consist of "bricks," which represent components of the SNN, with edges defining the flow of information. Fugu's core capabilities include an API for conventional programming environments, automated construction of graphical intermediate representations, and outputs to neural hardware compilers or its reference simulator. This makes it easier for users in machine learning and scientific computing to access and experiment with emerging neuromorphic hardware.

Fugu at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Static Synapse (No Learning)
Targeted Hardware: CPU, NIR
Source Code: github.com/sandialabs/Fugu

Genesis Research Notes

The GENESIS simulator, first referenced in 1994 and widely recognized in the neuroscience community, has accumulated over 1600 citations, including notable works such as HRLsim (2013), various versions of CARLsim, and Dendrify (2023). Its latest release, version 2.4, was published in 2019, indicating that while there has not been active development recently, its influence on spiking neural network (SNN) simulation is still profound. GENESIS was initially detailed in The Book of Genesis (1998, re-released in 2012), which has played a significant role in shaping the field over the past three decades, justifying its inclusion in this review despite the time gap since the last update.

Developed as a research tool to model biologically realistic neural systems, GENESIS allows simulations from sub-cellular components to large-scale neural networks. Its goal is to better understand the structure-function relationships in the nervous system and biology more broadly. Implemented in C and running on UNIX, GENESIS includes a parallel version, PGENESIS, for simulating large-scale models and conducting extensive parameter searches. By focusing on both single-cell models and large-scale network simulations, GENESIS provides a platform for collaborative research aimed at uncovering general principles of neuroscience organization, function, and computation.

Genesis at a Glance

Neuron Models: Hodgkin–Huxley, Multi-compartment
Learning Methods: Basic Hebbian
Targeted Hardware: CPU
Source Code: github.com/genesis-sim/genesis-2.4

GeNN Research Notes

The GeNN (GPU Enhanced Neuronal Network) simulator, widely popular in the past decade, serves as a GPU-accelerated backend for the Brian2 simulator through Brian2GeNN, enabling efficient simulations of large-scale spiking neural networks (SNNs). It supports a variety of pre-defined neuron and synapse models and offers extensive flexibility through Code-snippets that integrate seamlessly with Brian2-generated code. As a code generation framework, GeNN produces optimized CUDA code for both CPU and GPU, allowing for direct comparison between the two platforms. This design aims to speed up simulations significantly—up to 200 times faster on GPUs—especially for complex neuron models and sparse connectivity. Future development includes supporting multi-GPU setups to further enhance simulation performance.

Active development of GeNN continues, with the latest updates pushed to GitHub in October 2024. The simulator has been used successfully for training SNN classifiers on datasets like the Spiking Heidelberg Digits and Spiking Sequential MNIST, demonstrating its capability for real-time applications. GeNN allows users to control model dynamics without needing low-level GPU programming knowledge and supports a wide range of models, from simple Izhikevich neurons to complex Hodgkin-Huxley models. Its extensibility and ongoing collaborations with other simulators, like Brian2 and SpineML, ensure its continued relevance and usefulness in computational neuroscience research.

GeNN at a Glance

Neuron Models: Hodgkin–Huxley, Izhikevich, Equation-based, Kulkov Map
Learning Methods: Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, GPU
Source Code: github.com/genn-team/genn

jaxSNN Research Notes

JaxSNN is a novel library for event-based spiking neural network (SNN) simulation, distinguishing itself by using high-resolution event timestamps rather than discrete time-steps, which is typical of most SNN simulators. This enables the use of the EventProp algorithm for gradient-based training and supports hardware-in-the-loop for the forward pass. The library offers both fully software-based simulation and support for simulated neuromorphic hardware (mock) as well as real BSS-2 hardware. This unique approach addresses ongoing challenges in simulating event-driven networks, providing a significant step forward in integrating SNNs with modern machine learning frameworks.

The library is built on JAX and enables event-driven gradient estimation in analog neuromorphic hardware, specifically targeting systems like BrainScaleS-2. JaxSNN bridges traditional neuromorphic architectures and contemporary machine learning tools, offering flexibility in data structures and time management while retaining the Autograd functionality of JAX. This facilitates more efficient and adaptable training of spiking neural networks, enhancing their potential for real-world machine learning applications.

jaxSNN at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Static Synapse (No Learning), EventProp
Targeted Hardware: CPU, BrainScaleS
Source Code: github.com/electronicvisions/jaxsnn

Lava Research Notes

LAVA is a software framework designed for developing neuromorphic applications for the Loihi chip, providing library support for PyTorch to facilitate offline backpropagation training. It includes Lava-DL, which supports exporting models using the Neuromorphic Intermediate Representation (NIR) to enhance hardware compatibility. LAVA primarily supports LIF neurons, the basic type supported by Loihi hardware, and offers standard STDP functionality. The framework also provides access to additional variables and synaptic trace information, enabling the development of more complex STDP variants such as STDP-ET and Reward Modulated STDP.

This flexibility in modeling synaptic plasticity allows for more advanced neuromorphic learning mechanisms, expanding the range of potential applications for LAVA in both research and practical neuromorphic computing.

Lava at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, Loihi, NIR
Source Code: github.com/lava-nc/lava

Memristor-Spikelearn Research Notes

The Memristor-Spikelearn simulator is an open-source tool designed to study synaptic plasticity in spiking neural networks by incorporating detailed memristor and circuit models for more accurate simulations. It addresses the challenge that the change in conductance (delta G) of a memristor depends on both the voltage and the current conductance, which may not always match the desired learning pattern. The simulator emphasizes the importance of using detailed device models to ensure that simulations yield realistic results, helping to avoid misleading conclusions. The research also explores how circuit structures, such as 1R and 1T1R configurations, can impact the tradeoff between energy efficiency and accuracy in training and inference tasks.

One key concern highlighted by the authors is that Brian, a widely used simulator, lacks the necessary interface to incorporate device and circuit details into the model. Understanding the limitations of this interface is important, as overcoming these constraints could provide a significant tool for the research community. Possible solutions, such as reducing the size of conductance changes or identifying linear regions in the conductivity plot for better learning alignment, are proposed to mitigate issues with the memristor's conductance behavior.

Memristor-Spikelearn at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Reward-Modulated (M-STDP) (R-STDP)
Targeted Hardware: CPU
Source Code:
github.com/YLab-UChicago/Memristor-Spikelearn

N2S3 Research Notes

The N2S3 (Neural Network Scalable Spiking Simulation), also known as Nessy, simulator introduces a novel event-based architecture for spiking neural networks, leveraging artificial synapses through a "synapse box" that incorporates both volatile and non-volatile memristor devices to support short-term potentiation (STP) and long-term potentiation (LTP). Unlike traditional time-step-based simulators, N2S3 uses an event-driven approach to calculate spike currents and neuron membrane potentials, offering more flexibility and efficiency in simulating neural activity. The simulator shows promise for large-scale models, potentially extending beyond a single compute node, thanks to its use of Scala and the Akka actor library, which employs actors to simulate the exchange of spikes between neurons.

This simulator may be particularly valuable for research in the ENDS lab focusing on memristor-based STDP (spike-timing-dependent plasticity). The system has demonstrated better performance than traditional non-volatile artificial synapses in simulations, such as the MNIST handwritten digit recognition dataset. The study emphasizes the potential of neuromorphic systems and the synapse box to improve the energy efficiency and performance of neural networks, positioning it as a promising tool for future neural network architectures.

N2S3 at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Short-Term Plasticity (STP)
Targeted Hardware: CPU, Synapse Box
Source Code:
sourcesup.renater.fr/wiki/n2s3/downloads
git.renater.fr/anonscm/git/n2s3/n2s3.git

Nengo Research Notes

Nengo is a widely used simulator, with over 579 citations and significant ongoing development, including the latest release of version 4.0.0 in November 2023. Originally published in 2014, it has grown into a community-supported platform with over 10 related projects. Nengo employs the Neural Engineering Framework (NEF), a theoretical framework that allows modelers to conceptualize brain-like information processing by connecting simple neural models. The platform is designed for large networks of neurons, supporting both spiking and non-spiking models, and facilitates simulations of complex systems based on the NEF principles. The NengoLoihi extension enables model development for Loihi hardware, with version 1.1.0 released in 2022 for direct simulation of LIF models on Loihi.

Despite its robust development and functionality, Nengo may not align well with certain research areas, such as those involving hardware with unique dynamics. The absence of mechanisms for adding custom synapse or neuron models could limit its flexibility for some projects. However, Nengo's ability to import and export modules using the Neuromorphic Intermediate Representation (NIR) makes it compatible with diverse hardware platforms, further extending its applicability in neuromorphic computing.

Nengo at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich, Adaptive Integrate-and-Fire
Learning Methods: PES, RLS, BCM, OJA, VOJA, Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, GPU, SpiNNaker, FPGA, NIR, Loihi
Source Code:
www.nengo.ai
github.com/nengo/nengo-loihi

NengoDL Research Notes

NengoDL is an innovative framework that bridges the gap between standard TensorFlow deep neural networks (DNNs) and Nengo spiking neural networks (SNNs). It achieves this by using a TensorFlow-based simulation backend, enabling users to define models through the Nengo frontend while supporting the neuron models and learning mechanisms of the standard Nengo simulator. NengoDL allows the integration of TensorFlow's learning methods into Nengo models, although challenges arise due to the non-differentiability of many neuromorphic models, like the Leaky Integrate-and-Fire (LIF) neuron. To address this, NengoDL uses a rate-based approximation for spiking neurons, allowing gradient-based optimization techniques to be applied through surrogate gradient descent.

One of NengoDL's key features is its ability to combine DNNs and SNNs within the same model, such as processing images with a convolutional neural network (CNN) and passing the output to

an SNN. It takes advantage of TensorFlow's hardware support for both CPU and GPU, as well as its performance optimizations. While NengoDL shows great promise for integrating deep learning and neuromorphic models, its limitations in terms of flexibility for custom neuron dynamics and the need for custom neurons to be supported by the NengoDL backend make it less suitable for highly specialized simulation needs, such as those in the ENDS lab.

NengoDL at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich, Adaptive Integrate-and-Fire
Learning Methods: PES, RLS, BCM, OJA, VOJA, Surrogate Gradient Back Propogation, Standard Error Back Propogation, ANN-to-SNN Conversion
Targeted Hardware: CPU, GPU
Source Code: github.com/nengo/nengo-dl

NEST Research Notes

The NEST (NEural Simulation Tool) simulator has been under active development for over two decades, with the latest release, version 3.8, launched in July 2024, demonstrating its ongoing research and development activity. It supports the NESTML description language and uses differential equations to describe both neuron behaviors and learning rules, offering flexibility in constructing spiking neural networks (SNNs). NEST provides pre-configured objects for creating networks, allowing users to instantiate models with desired parameters. The simulator uses an event-driven approach, which optimizes communication efficiency in systems where element evaluation is expensive. However, in large, highly connected networks, the overhead of event handling can become significant, making time-driven strategies more efficient in such cases.

NEST is designed for simulating large, structured neuronal systems and is intended to balance the biological detail of individual neurons with the scale required for large networks. It supports over 50 neuron models and nearly 30 synapse models, with variations of basic neuron and synapse types. The framework employs an object-oriented design in C, breaking down the system into components represented by classes. These components share a common interface but vary in functionality, allowing dynamic configuration of complex neuronal networks. Although the original 2002 NEST paper offers insights into the initial implementation, up-to-date information on neuron models, learning methods, and encoding mechanisms can be found in the User Guide. The high degree of development and research activity surrounding NEST justifies its inclusion in this simulator review.

NEST at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich, Adaptive Integrate-and-Fire, Hodgkin–Huxley, Multi-compartment, Adaptive Exponential Integrate-and-Fire (aEIF), Equation-based
Learning Methods: Spike Timing Dependant Plasticity (STDP), Basic Hebbian, Static Synapse (No Learning), Short-Term Plasticity (STP), Triplet STDP, EventProp, Equation-based
Targeted Hardware: CPU
Source Code:
www.nest-simulator.org/
github.com/nest/nest-simulator

NEST GPU Research Notes

NEST GPU, formerly known as NeuronGPU, is a GPU library designed for large-scale simulations of spiking neural networks. It supports LIF and AdEx neuron models, along with various synapses, spike generators, and recording tools, delivering high efficiency in simulating cortical microcircuit models and balanced networks. However, NEST GPU does not support multi-compartment models and only includes the STDP learning mechanism. While it allows neuron models to be defined using differential equations, there is no user interface for this, requiring modifications to the underlying code to add new neuron model definitions.

The simulator employs a parallel implementation of the fifth-order Runge-Kutta method with adaptive step-size control for solving differential equations, ensuring high performance in large-scale spiking neural network simulations, particularly with AdEx models and conductance-based synapses. NEST GPU's computational efficiency is further enhanced by its spike delivery algorithm, which optimizes spike propagation and buffer updates, resulting in superior simulation speeds compared to other simulators like GeNN and NEST. With its Python interface and a wide range of connection rules and parameter distributions, NEST GPU offers ease of use and achieves simulation speeds close to biological real-time, making it a promising tool for large-scale, high-performance neural network simulations.

NEST GPU at a Glance

Neuron Models: Leaky Integrate-and-Fire, Adaptive Exponential Integrate-and-Fire (aEIF), Izhikevich, Equation-based
Learning Methods: Spike Timing Dependant Plasticity (STDP), Equation-based
Targeted Hardware: GPU
Source Code: github.com/nest/nest-gpu

Neuron Research Notes

The Neuron simulator, first published in 2006, is actively developed with the latest release, version 8.2.6, launched in July 2024. With over 2000 citations on Google Scholar and a significant number of publications listed on its website, Neuron continues to play an important role in neuroscience research. This activity, combined with its latest updates, justifies its inclusion in this review. Neuron is primarily designed for fine-grained neuron modeling, including complex dendritic tree models, and supports the NMODL description language. It targets the neuroscience community, particularly those focused on detailed modeling, rather than deep learning applications.

The simulator uses specialized equation solvers and modeling techniques suited to the unique needs of dendritic structures, which sets it apart from other simulators. While it provides comprehensive modeling capabilities, Neuron is geared toward advanced neuroscience research, and its documentation can be difficult to follow for newcomers. Much of the information on supported neuron models and learning mechanisms comes from external papers rather than the official Neuron documentation, making it challenging for users to quickly understand how to get started with the simulator.

Neuron at a Glance

Neuron Models: Leaky Integrate-and-Fire, Hodgkin–Huxley, Integrate-and-Fire, Multi-compartment
Learning Methods: Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU
Source Code: github.com/neuronsimulator/nrn

NeuroPack Research Notes

NeuroPack is an algorithmic-level simulator specifically designed for modeling memristor-based synapses in spiking neural networks (SNNs). Unlike traditional simulators that focus on circuit-level simulations, NeuroPack addresses the challenge of determining whether a memristor synapse can perform specific online or offline learning tasks. The simulator maps weights linearly to memristor resistances, but due to the non-linear nature of memristors, achieving precise weight changes is difficult. To address this, NeuroPack includes a module for calculating the necessary biasing parameters to control the desired weight changes. It has been applied to tasks such as MNIST classification, demonstrating its potential for validating the effectiveness of memristor-based synapses in SNNs.

Although NeuroPack does not support hardware-in-the-loop configurations with real memristors, it provides a flexible Python-based platform that allows researchers to experiment with different neuron models, learning rules, and memristor models. The simulator is capable of configuring multi-layer SNNs with Winner-Takes-All (WTA) mechanisms and utilizing Leaky Integrate-and-Fire (LIF) neurons. This makes it an excellent tool for investigating memristor neuro-inspired architectures, particularly in tasks involving online learning and offline classification, offering promising avenues for future research in the area of memristor-based neural networks.

NeuroPack at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich
Learning Methods: Spike Timing Dependant Plasticity (STDP), Surrogate Gradient Back Propogation
Targeted Hardware: CPU
Source Code: github.com/hjq310/NeuroPack

NEUTRAMS Research Notes

NEUTRAMS is a toolkit designed to facilitate the deployment of spiking neural networks (SNNs) and artificial neural networks (ANNs) on specialized neuromorphic hardware. The toolkit includes a simulator that allows various network models to be tested under the constraints of the target hardware. However, it is important to note that NEUTRAMS is intended for inference-only configurations, where models must first be trained on other platforms like Nengo or BRIAN before being transformed for use with NEUTRAMS. The system includes a neural network transformation algorithm, a clock-driven simulator for neuromorphic chips, and an optimized runtime tool to map neural networks onto hardware for efficient resource utilization.

One of the key strengths of NEUTRAMS is its focus on challenges specific to neuromorphic hardware, particularly crossbar-structured memristors, which is relevant to research in the ENDS lab. The toolkit has been validated on real neuromorphic chips and processor-in-memory architecture designs for ANNs. Additionally, the authors provide comparative results between different types of neural networks, offering valuable insights into optimizing neuromorphic architecture designs.

NEUTRAMS at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: ANN-to-SNN Conversion, Static Synapse (No Learning)
Targeted Hardware: CPU, TIANJI, PRIME
Source Code: github.com/hoangt/neutrams

Norse Research Notes

Norse is a deep learning framework built on top of PyTorch, designed to support the import and export of modules using the Neuromorphic Intermediate Representation (NIR). It provides good documentation, along with examples and tutorials, making it accessible for users to get started. While Norse does not appear to support specific neuromorphic hardware backends, its design makes it flexible for implementing a variety of neuron models, particularly LIF (Leaky Integrate-and-Fire) variants, of which it offers a substantial range.

This framework is ideal for those working in deep learning and neuromorphic computing, offering easy-to-use tools to create and manipulate neural models. However, its lack of direct hardware support limits its application to simulations and research on software-defined neural networks rather than hardware-accelerated systems.

Norse at a Glance

Neuron Models: Leaky Integrate-and-Fire, Integrate-and-Fire, Izhikevich, Recurrent LIF
Learning Methods: Surrogate Gradient Back Propogation, Back Propogation Through Time (BPTT)
Targeted Hardware: NIR, CPU, GPU
Source Code:
norse.github.io/norse
github.com/norse/norse

PymoNNto Research Notes

PymoNNto(rch) is a versatile simulator framework that supports a variety of neuron models, including LIF, Izhikevich, and Hodgkin-Huxley. It includes learning mechanisms such as STDP, intrinsic neuron plasticity, and evolutionary parameter tuning, as well as features like synaptic weight normalization, refractory periods, and NOX diffusion-based homeostasis to stabilize learning. The key distinction between PymoNNto and PymoNNto(rch) is that the latter is built using PyTorch, which enables GPU support and the use of modules across any PyTorch-compatible hardware, offering greater flexibility and efficiency compared to the NumPy-based PymoNNto.

Designed with a discrete time-based simulation framework, PymoNNto(rch) updates the system state based on the last time step, making it suitable for various brain-inspired neural network designs. The toolbox's modular structure allows the creation of custom neuron and synapse behaviors through Network classes, NeuronGroups, and SynapseGroups, and it includes a graphical user interface for real-time observation and modifications. Additionally, PymoNNto supports integration with libraries like TensorFlow and Cython for optimized performance, making it a flexible and efficient tool for designing and analyzing neural networks.

PymoNNto at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich, Diesmann, Brunel and Hakim, Wang and Buzsáki, Hindmarsh and Rose, Hopfield, Hodgkin–Huxley
Learning Methods: Spike Timing Dependant Plasticity (STDP), Synaptic Weight normalization, Intrinsic Plasticity, Refractory Period, NOX diffusion-based homeostasis, Evolutionary Parameter Tuning
Targeted Hardware: GPU, CPU
Source Code: github.com/trieschlab/PymoNNto

PymoNNtorch Research Notes

PymoNNto(rch) is a highly flexible simulator framework that supports various neuron models, including LIF, Izhikevich, and Hodgkin-Huxley, and integrates learning mechanisms such as STDP, intrinsic neuron plasticity, and evolutionary parameter tuning. It also stabilizes learning through features like synaptic weight normalization, refractory periods, and NOX diffusion-based homeostasis. PymoNNto is built using NumPy, while PymoNNtorch is implemented with PyTorch, allowing it to leverage GPU acceleration and enabling module development across any GPU supported by PyTorch, rather than relying on hardware-specific implementations.

The simulator operates within a discrete time-based framework, where simulation time advances step-by-step and updates the system state based on the most recent time step. PymoNNto(rch) is designed for modularity and flexibility, making it suitable for custom SNN models. The framework shows significant performance improvements compared to other simulators like NEST, ANNarchy, and Brian in certain cases, particularly through the use of PyTorch's optimized operations and GPU support. The paper compares network implementations across these simulators, showcasing how PymoNNto(rch) outperforms others in speed and efficiency for various spiking neural network models.

PymoNNtorch at a Glance

Neuron Models: Leaky Integrate-and-Fire, Izhikevich, Diesmann, Brunel and Hakim, Wang and Buzsáki, Hindmarsh and Rose, Hopfield, Hodgkin–Huxley
Learning Methods: Spike Timing Dependant Plasticity (STDP), Synaptic Weight normalization, Intrinsic Plasticity, Refractory Period, NOX diffusion-based homeostasis, Evolutionary Parameter Tuning
Targeted Hardware: GPU, CPU
Source Code: github.com/cnrl/PymoNNtorch

PySNN Research Notes

PySNN is an SNN simulator built on top of PyTorch, but it has minimal available documentation, with most of the information coming from the code comments. Despite its limited public references, one notable mention is in the paper Neuromorphic Control for Optic-Flow-Based Landing of MAVs Using the Loihi Processor by Dupeyroux et al. (2021), where PySNN was used alongside an evolutionary technique to train an SNN. The trained network was then deployed on the Loihi processor for the task of controlling the landing of a micro aerial vehicle (MAV) based on optic flow.

PySNN at a Glance

Neuron Models: Leaky Integrate-and-Fire, Integrate-and-Fire, Adaptive Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Reward-Modulated Eligibility Trace (M-STDP-ET) (R-STDP)
Targeted Hardware: GPU, CPU
Source Code: github.com/BasBuller/PySNN

RANC Research Notes

RANC is a versatile toolkit that integrates both software simulation and hardware emulation for Spiking Neural Networks (SNNs), providing both a software simulator and an FPGA-based simulator. TensorFlow is used for model definition and training, with subsequent preparation for RANC

simulation via libraries that interface with TensorFlow. One key constraint is the requirement to use synaptic connection probabilities rather than weights, enabling the use of backpropagation for training. However, for forward propagation, these probabilities must be converted to binary values (0 or 1). The system offers a high degree of customization in neuron behavior through configuration parameters, though changes in network topology may require significant modifications to the simulators.

RANC demonstrates its flexibility by emulating the IBM TrueNorth neuromorphic processor and validating both software and hardware simulators using the MNIST and EEG datasets. The toolkit supports trend-based analysis and architectural impact studies, with ongoing work to enhance its features, such as multicast routing, spike-timing dependent plasticity, and heterogeneous core configurations. A scalability analysis conducted on the Alveo U250 FPGA shows linear scaling with a core frequency of 177.336 MHz, supporting up to 259,072 distinct neurons and 73.3 million synapses, making RANC a powerful tool for neuromorphic computing research and development.

RANC at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Synaptic Connection Probability Back Propogation
Targeted Hardware: CPU, FPGA
Source Code: github.com/UA-RCL/RANC

RAVSim Research Notes

RAVSim is an interactive simulator for Spiking Neural Networks (SNNs) built on the LabVIEW platform, which requires a paid subscription, though it is unclear whether the freely available LabVIEW runtime engine can run RAVSim. The simulator uses a combination of clustering and winner-take-all mechanisms for unsupervised learning but provides limited clarity on its actual capabilities. While the paper claims RAVSim can process full RGB images, it contradicts itself by stating that it currently handles only binary images (with two levels of intensity), suggesting that the system can only process black-and-white images, not full RGB data as implied.

The research paper highlights RAVSim's capabilities for image classification tasks, such as achieving 91.8

RAVSim at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods:
Targeted Hardware: CPU
Source Code: github.com/Rao-Sanaullah/RAVSim

SaRNN Research Notes

The SaRNN simulator implements Spiking Neural Networks (SNNs) as TensorFlow Recurrent Neural Networks (RNNs), compiling them into a static computation graph for improved performance and efficiency. It uses MPI for communication, enabling simulations to run on large compute clusters, although there is no mention of GPU support. The simulator is intended for inference only, employing an ANN-to-SNN conversion technique for training. The model relies on a NonLeaky Integrate and Fire (NL-LAF) neuron, which resets the membrane potential after a spike through subtraction, making it similar to a basic Integrate and Fire (IF) neuron. The paper also discusses the mapping of their SNN model to real SpiNNaker hardware, providing some performance results.

In addition to the implementation details, the research emphasizes the potential of SNNs for computer vision applications, offering a smooth tradeoff between latency, power, and accuracy. The study introduces novel optimization techniques to reduce latency and power consumption by 1-2 orders of magnitude in converted SNNs. Furthermore, Pareto metrics are proposed to evaluate and optimize models based on accuracy-time or accuracy-power tradeoff curves, facilitating better model optimization for improved latency and power efficiency in ANN-converted SNNs.

SaRNN at a Glance

Neuron Models: Integrate-and-Fire
Learning Methods: ANN-to-SNN Conversion
Targeted Hardware: CPU, SpiNNaker
Source Code: github.com/WISION-Lab/sarnn

SINABS Research Notes

The Sinabs library, built on PyTorch, is a deep learning tool for Spiking Neural Networks (SNNs) that supports ANN-to-SNN conversion, Back Propagation Through Time, Surrogate Gradient functions, and custom neuron and synapse models. It also simplifies the transfer of model parameters and the generation of configurations compatible with the SPECK neuromorphic hardware platform. This makes it a promising resource for exploring energy-efficient machine intelligence and custom SNN applications, especially when integrated with SPECK's capabilities.

SPECK itself is a neuromorphic system co-designed to achieve energy-efficient computation by mimicking the human brain's neurons and synapses. It features an asynchronous design with low resting power, allowing for dynamic energy consumption based on input activity. The platform addresses challenges like the dynamic imbalance in SNNs and integrates high-level brain mechanisms, such as attention, to further enhance efficiency. The research emphasizes dynamic computing, sparse sensing, and event-driven computation, highlighting SPECK's potential for low-latency, low-power edge computing applications.

SINABS at a Glance

Neuron Models: Integrate-and-Fire, Leaky Integrate-and-Fire
Learning Methods: ANN-to-SNN Conversion, Surrogate Gradient Back Propogation, Back Propogation Through Time (BPTT)
Targeted Hardware: CPU, Speck
Source Code: github.com/synsense/sinabs

SNNToolbox Research Notes

This toolbox is designed to transform models written in Keras, Lasagne, and Caffe into Spiking Neural Networks (SNNs). It includes built-in simulation capabilities and can export models to platforms like Brian2 and PyNN. The focus of the research is on converting deep Convolutional Neural Networks (CNNs) into SNNs by introducing spiking equivalents of common operations, such as max-pooling, softmax, batch normalization, and Inception modules.

The study demonstrates the conversion of popular CNN architectures like VGG-16 and Inception-v3 into SNNs, achieving strong results on datasets like MNIST, CIFAR-10, and ImageNet. This showcases the potential of SNNs to maintain classification accuracy while reducing operations, which is particularly beneficial for power-efficient neuromorphic chips and embedded applications. The research also explores methods to reduce the classification error rate of deep SNNs, highlighting the balance between error rate and operation efficiency in these models.

SNNToolbox at a Glance

Neuron Models: Integrate-and-Fire
Learning Methods: ANN-to-SNN Conversion
Targeted Hardware: CPU
Source Code:
github.com/NeuromorphicProcessorProject/snn_toolbox

snnTorch Research Notes

snnTorch, based on PyTorch, supports importing and exporting modules using the Neuromorphic Intermediate Representation (NIR) and provides a robust framework for building and training Spiking Neural Networks (SNNs). The associated publication offers an excellent overview of the state of SNNs as of 2023 and is accompanied by high-quality tutorials that guide users through the process of building and training SNNs, along with various features of the snnTorch simulator. While the official documentation does not specifically mention Spike-Timing Dependent Plasticity (STDP) as a supported learning mechanism, there are code references to STDP in the repository, suggesting it may be released as a future feature.

The research paper titled Training Spiking Neural Networks Using Lessons From Deep Learning explores applying insights from deep learning, gradient descent, backpropagation, and neuroscience to biologically plausible spiking neural networks. This provides valuable guidance for improving the training of SNNs, focusing on techniques such as surrogate gradient descent for training these networks.

snnTorch at a Glance

Neuron Models: Leaky Integrate-and-Fire, Recurrent LIF, Lapicque, Recurrent Synaptic, Synaptic, Alpha, SLSTM
Learning Methods: Surrogate Gradient Back Propogation, Back Propogation Through Time (BPTT)
Targeted Hardware: CPU, GPU, NIR, Graphcore Intelligence Processing Unit (IPU)
Source Code: github.com/jeshraghian/snntorch

SPAIC Research Notes

SPAIC is a Python-based framework that bridges Deep Neural Network (DNN) tools and computational neuroscience simulators, enabling researchers to leverage the bio-realism of neuroscience-based simulators while benefiting from the performance characteristics of DNN simulators. It facilitates hybrid neural networks by allowing a Spiking Neural Network (SNN) to be trained using Stochastic Gradient Descent (SGD) and then enhanced with Spike-Timing Dependent Plasticity (STDP). SPAIC supports various backends, including PyTorch, TensorFlow, JAX, and Darwin, offering flexibility and efficiency in simulating and training brain-inspired models.

The framework includes a well-structured related works section that details the features of popular SNN simulators and the application areas they target. SPAIC provides tools for building customizable network models through Assembly, Connection, and Learner objects, supporting diverse connection forms and synapse types. As a tool under continuous development, SPAIC offers a user-friendly, flexible, and high-performance platform for constructing and testing brain-inspired models, aimed at advancing AI research.

SPAIC at a Glance

Neuron Models: Leaky Integrate-and-Fire, Hodgkin–Huxley, Izhikevich, Adaptive Exponential Integrate-and-Fire (aEIF)
Learning Methods: Spike Timing Dependant Plasticity (STDP), Surrogate Gradient Back Propogation, Reward-Modulated (M-STDP) (R-STDP), FORCE Training (Reservoir Computing), SpikeProp
Targeted Hardware: CPU, GPU
Source Code: github.com/zju-bmi-lab/SPAIC

SPIKE Research Notes

Spike is a GPU-optimized Spiking Neural Network (SNN) simulator designed to demonstrate the power of GPU-based simulation. It focuses on performance enhancements through optimizations like timestep grouping, active synapse grouping, and delay insensitivity, which help improve simulation speed, especially in large-scale networks. Spike performs well compared to both GeNN (another GPU-based simulator) and CPU-based simulators, showing superior performance without slowing down as synaptic populations increase or delay structures are added. The simulator, however, does not support a wide variety of neuron models or learning mechanisms and appears to be primarily aimed at showcasing GPU-enhanced SNN simulation rather than offering a comprehensive tool for neuromorphic research.

While documentation for Spike is available, much of the most useful content is hosted on a Google Site linked in the GitHub repository. The research paper also includes benchmark comparisons between Spike and other simulators, with a public repository containing code and iPython notebooks for visualizing and analyzing network behavior across different platforms.

SPIKE at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Static Synapse (No Learning)
Targeted Hardware: GPU
Source Code: github.com/OFTNAI/Spike

SpikingJelly Research Notes

SpikingJelly is an open-source machine learning platform built on PyTorch, designed to facilitate deep learning with Spiking Neural Networks (SNNs). It aims to address challenges in adapting existing SNN simulators for deep learning applications, which often require extensive knowledge and time due to inconsistent programming languages, coding styles, and neuron definitions. By providing a standardized framework based on a widely-used Artificial Neural Network (ANN) platform, SpikingJelly makes it easier for researchers to explore the potential benefits of SNNs for deep learning tasks. The platform supports supervised learning methods like surrogate gradient descent with backpropagation through time and ANN-to-SNN conversion, the latter being based on rate coding which requires more time steps than the surrogate gradient approach.

SpikingJelly accelerates the training of deep SNNs by 11x, offering flexibility for custom models and supporting multilevel inheritance and semiautomatic code generation. The platform enables efficient deployment of SNNs on neuromorphic chips and enhances performance in real-world applications, from basic dataset classification to human-level tasks. It provides tools for preprocessing neuromorphic datasets, building deep SNNs, optimizing parameters, and deploying them on

neuromorphic hardware. The framework also includes modules for spiking neurons, layer-by-layer propagation, and CUDA code generation, contributing to the growing field of spiking deep learning.

SpikingJelly at a Glance

Neuron Models: Leaky Integrate-and-Fire, Integrate-and-Fire
Learning Methods: Surrogate Gradient Back Propogation, Back Propogation Through Time (BPTT), ANN-to-SNN Conversion, Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: CPU, GPU, Loihi, Tianjic Lynxi KA200
Source Code: github.com/fangwei123456/spikingjelly

SpykeTorch Research Notes

SpykeTorch is an open-source Spiking Neural Network (SNN) simulator built on PyTorch, optimized for convolutional SNNs that use one spike per neuron, utilizing Time to First Spike encoding. The framework implements learning rules like STDP and R-STDP and can reproduce various study results by leveraging PyTorch's computational functions for just-in-time optimization across different platforms. SpykeTorch supports tensor-based computations and is designed to gradually extend its functionalities, with future plans for batch processing, improved speed, and multi-modal SNN models for applications like auditory systems.

In addition to its core capabilities, SpykeTorch includes features like lateral inhibition, local normalization, and winners-take-all competition, which contribute to plasticity and decision-making processes in SNNs. The framework's support for time-to-first-spike coding, convolutional layers with STDP, and integration within the PyTorch ecosystem makes it a flexible and scalable tool for SNN simulations, with plans for further enhancement.

SpykeTorch at a Glance

Neuron Models: Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP), Reward-Modulated (M-STDP) (R-STDP)
Targeted Hardware: CPU, GPU, GPU Cluster
Source Code: github.com/miladmozafari/SpykeTorch

Spyx Research Notes

Spyx is a JAX-based library designed for the simulation and optimization of Spiking Neural Networks (SNNs), with a focus on energy-efficient temporally-sparse computations. It treats SNNs as recurrent neural networks (RNNs) using DeepMind's Haiku library, allowing for efficient training and deployment of deep neural networks. Spyx leverages Just-In-Time (JIT) compilation to optimize SNNs, achieving performance comparable to custom CUDA implementations while maintaining flexibility and interoperability with PyTorch-based frameworks. The library also supports user-defined surrogate gradient functions and neuron models with minimal code, making it adaptable for various applications. It is compatible with GPUs and TPUs, although TPU support is not fully functional yet due to library version issues, though it is expected to be available soon.

A novel feature of Spyx is its use of the Neuromorphic Intermediate Representation (NIR) for serializing models, making them portable and allowing for easy transfer between platforms. The library maximizes JIT compilation opportunities and follows functional programming principles, enabling efficient training of SNNs in JAX. Spyx interfaces seamlessly with other JAX libraries, enhancing its functionality and compatibility. This design makes it a flexible and powerful tool for simulating and optimizing SNNs while maintaining energy efficiency and computational performance.

Spyx at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Surrogate Gradient Back Propogation
Targeted Hardware: GPU, NIR
Source Code: github.com/kmheckel/spyx

SuperNeuro Research Notes

SuperNeuro is a high-performance neuromorphic simulator developed by Oak Ridge National Laboratory, offering two distinct simulation backends. The first, SuperNeuroMAT, utilizes matrix computation for homogeneous simulations with LIF neurons and supports STDP, but runs only on CPUs. The second, SuperNeuroABT, employs agent-based modeling to support heterogeneous neuron types, including LIF, and operates on GPUs, also supporting STDP. Designed for efficiency rather than biological realism, SuperNeuro is capable of simulating large-scale networks quickly, such as a 100,000-neuron brain model in just five minutes on MAT mode, significantly outperforming other simulators like NEST and Brian2.

While SuperNeuro is not specifically tailored for neuroscience or deep learning, it is geared towards the rapid development of neuromorphic algorithms, including both spiking and non-spiking models, as seen in applications like efficient neuromorphic computing. Despite its strong performance and scalability, the simulator lacks comprehensive documentation and user guides, making it challenging for researchers to determine its suitability for specific projects. The available tutorials are brief, and the absence of detailed guidance limits its accessibility to users unfamiliar with the system.

SuperNeuro at a Glance

Neuron Models: Leaky Integrate-and-Fire
Learning Methods: Spike Timing Dependant Plasticity (STDP)
Targeted Hardware: GPU, CPU
Source Code:
github.com/ORNL/superneuro
github.com/ORNL/superneuromat
github.com/ORNL/superneuroabm