Comparing SNN Simulators: BindsNET and NENGO

ALSTON CHEN, Amherst College, USA

ABSTRACT Spiking Neural Network (SNN) simulators like Nengo and BindsNET play crucial roles in developing and understanding neuromorphic computing systems. Nengo and BindsNET differ significantly in their design philosophy and functionality, particularly regarding visualization capabilities and hardware flexibility. BindsNET offers a flexible environment where simulation speed on CPUs significantly outperforms GPUs in specific configurations, suggesting that the choice of hardware does not critically impact its performance. Conversely, Nengo operates within a web browser, limiting the ability to switch computational resources but simplifying access and use. This paper explores these differences in detail, evaluating each simulator's strengths and limitations, primarily in Windows, to guide users in selecting the appropriate tool based on their specific requirements in neuromorphic simulation.

ACM Reference Format:
Alston Chen. 2024. Comparing SNN Simulators: BindsNET and NENGO. 1, 1 (May 2024), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As technology becomes more and more advanced, the prospect of creating a brain-like computer becomes more and more likely. With the development of Spiking Neural Network (SNN), a form of neuromorphic computing, and specific hardware, the promise of more energy-efficient and more accurate artificial intelligence models can be realized thanks to the composition and format of how the brain works. Yet, the concept of such networks is difficult to grasp without visualization, especially with the difficulty of applying, developing, and training SNNs. This is important as this technology shows immense potential for the future of neural networks. Not only does it significantly reduce power consumption, but it has the potential to achieve relatively similar accuracy compared to modern ANNs. [axn 2015][ben 2019][dee 2020]

Firstly, it is essential to understand what precisely an SNN is. Generally, neural networks use standard activation functions, including ReLU (Rectified Linear Unit), sigmoid, and tanh. However, SNNs communicate by generating discrete events (spikes) when their membrane potential—an internal state—exceeds a certain threshold. The information is encoded in the timing or frequency of these spikes, and because SNNs utilize spikes, the energy cost for training and use can be significantly reduced. This is because of the inherent property of SNNs to only fire when the spikes pass a certain threshold. However, ANNs operate asynchronously, meaning that all neurons update simultaneously, leading to much more power consumed. It should also be noted that SNNs emulate the brain's neuron functions: The human brain has a total budget of 20W, while current technology for recognizing 100 objects takes 250W. [mac [n. d.]]

However, even though Neuromorphic computing can be highly energy efficient and can be utilized in both formal and event-driven networks, SNNs are also greatly dependent on their hardware components to be energy efficient, to the point that the bottleneck of SNNs is silicon technology, though that hasn't reached that yet. And the current goal is to have artificial intelligence inside personal mobile devices. Still, typically, devices with limited battery life need to connect to the cloud for processing on more powerful machines, even when running SNNs. Therefore, there's a critical need for co-designing low-power hardware and algorithms to expand the practical uses of deep learning on SNNs. This creates a need for expert knowledge in neuroscience and computer science to understand how SNNs are run and to begin researching and developing your SNNs.

With this in mind, many simulators created to visualize SNNs emerged, addressing the difficulties in understanding SNNs by non-experts. Two of the most popular simulators that include visualizations for training and modeling SNNs are Nengo and BindsNET:

1)Nengo, currently based in Python, offers an extensive library for creating graphs and plots within its GUI browser system, allowing users to create data visualizations of the datasets they hope to load and explore. The first version of Nengo was released in 2003 in Java, and with significant changes being made, the simulation eventually turned to a Python interface instead. Nengo also offers examples in the startup packages such as...

2)BindsNET, built on top of the PyTorch library, simulates the training of SNNs through a series of windows showing the different graphs that users can create. The trained network can then be tested, receiving the accuracy of the trained network in the form of weighting and proportional accuracy. The base BindsNET model comes with an MNIST dataset example with more on their GitHub page, such as TensorFlow and dot tracing.

Many other simulators that have been created include GenericSNN, SuperNeuro, and EvtSNN. These simulators have also been developed for visualization and offer real-time simulating of SNNs while utilizing their respective libraries and addressing issues of use by non-experts. Further research can be done to understand the differences in all of these simulators, but for the scope of this paper, BindsNET and Nengo will be the primary focus.

During this research period, another tool that was investigated was LAVA, an open-source software library dedicated to developing algorithms for neuromorphic computation. Section 6.4 will touch upon the problems that arose during installation and usage.

Motivated by the topic of neuromorphic computing and the prospects of creating a way to simplify the concept of SNNs, this paper will focus on the individual components of both Nengo and BindsNET, comparing the visuals, the libraries, and the ease of use to find what can be improved and understand how each works as an SNN simulator.

Author's Contact Information: Alston Chen, Amherst College, Amherst College, USA, Achen26@amherst.edu.

ACM XXXX-XXXX/2024/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SPIKING NEURAL NETWORKS (SNNS)

$$\frac{dV}{dt} = C1\left(I - \frac{V}{R}\right) \tag{1}$$

V is the membrane potential. C is the membrane capacitance. R is the membrane resistance. I am the input current.

As mentioned in Section 1, spiking neural networks use discrete events to identify when to fire a neuron. But what exactly does that mean? The model must first receive data that is then encoded to be inputted into the model. For example, when loading a widespread neural network training set like MNIST, the images must first be encoded in a way that can be read and inputted into the model. Some of these encoding schemes include[spa 2021]:

1)Rate Encoding: The spikes' frequency represents the input signal's intensity or magnitude.

2)Temporal Encoding: The timing of the spikes conveys information, with the exact moments of firing carrying significance.

3)Population Encoding: Multiple neurons represent the input, with the pattern of activity across this group encoding the signal.

There are then different neurons for transmission. While presynaptic neurons transmit the signal toward a synapse, a postsynaptic neuron transmits the signal away from the synapse. Each neuron has its own membrane potential, and when that membrane threshold is crossed, it fires. Figure 1 shows an example of a possible activation function an SNN may utilize for its membrane threshold. This is the activation function of an Integrate and Fire Model, and this model effectively simulates the basic "all-or-nothing" spiking behavior of biological neurons, where:

1)Integration: The neuron integrates incoming spikes, adding to the membrane potential.

2)Fire: When the membrane potential exceeds a certain threshold, the neuron fires a spike and resets its potential to a baseline level.

This is the process of the pre-synaptic neuron. After the threshold is crossed, the spike ripples throughout the network, increasing the weight of each neuron depending on the distance. The post-synaptic neurons will also evaluate the pre-synaptic neurons to determine whether they should fire.

The learning process can then differ based on different learning rules. One of the most common post-synaptic synaptic weights is adjusted based on the relative timing of spikes between pre-synaptic and post-synaptic neurons. If the pre-synaptic neuron fires a spike just before the post-synaptic neuron, then the synapse connecting them is strengthened (Long-Term Potentiation or LTP). Conversely, if the pre-synaptic synaptic neuron fires after, the post-synaptic synapse is weakened (Long-Term Depression or LTD).

$$\text{For LTP}(\Delta t > 0) : \Delta w = A^{+} \exp(\frac{\Delta t}{-\tau^{+}})) \tag{2}$$

$$\text{For LTD}(\Delta t < 0) : \Delta w = -A^{-} \exp(\frac{-\Delta t}{\tau^{-}}) \tag{3}$$

Figure 2a shows the mathematical formulation of STDP for when the synapse is strengthened where A+ is the maximum amount of potentiation and τ- is the time constant for the decay of the potentiation effect as the interval increases.

Figure 2b shows the mathematical formulation of STDP for when the synapse is weakened where A- is the maximum amount of depression and τ- is the time constant for the decay of the depression effect as the interval increases.

The final stage of this process is interpreting the spikes emitted by the output neurons of the network to make them useful for real-world applications such as pattern recognition or decision-making tasks. Some methods of interpretation include:

Rate firing: measuring the firing rate of each neuron over a fixed interval. The rate can directly correspond to features of the input data or the desired output values.

Temporal Decoding: interpreting the precise timings of spikes. This can be especially useful in tasks where timing is critical, such as dynamic decision-making or processing time-series inputs.

Population Decoding: The pattern of spikes across a group of neurons determines the output. This method can increase the robustness and accuracy of the interpretation.

Now, that is the simplified process of how an SNN is trained and deployed without going into the specifics of the hardware components, and for this paper, the hardware takes a back seat. Simulators for these processes are the main focus. So, what does this look like when developing and using SNNs?

3 SIMULATING SNNS: BINDSNET

Simulators provide a direct pathway to understanding the basics of SNNs and the development of one's SNNs when visualizing and using them. First, let's take a look at the process of using BindsNET:

1) The installation process is streamlined with pip and BindsNET's git repository, a simple command in the terminal: pip install git+https://github.com/BindsNET/bindsnet.git

2)BindsNET runs on Python, allowing easy access to the base code used for simulating the training and testing process of the MNIST dataset.

3) you must set up your Python environment properly before creating your simulations. This involves creating a virtual environment for your project to manage dependencies and avoid conflicts. For instance, using `venv` or `conda` can help encapsulate all your project-specific packages neatly.

4) BindsNET documentation includes creating the network for simulating and training your simulations.

5)In gathering data and seeing output, a monitor method is embedded in the BindsNET network. The monitor lets you see the output layer spikes and voltages.[bin 2018] [spi 2019][spi 2019]

3.1 BindsNET MNIST

As stated before, the base example shown in BindsNET works with MNIST and can be run on a local machine without much trouble. The library allows you to select your CPU or GPU depending on whether a GPU is available. However, this doesn't seem to affect the performance much as training speeds for MNIST remain relatively similar regardless of the device chosen. With BindsNET's MNIST training example that uses unsupervised learning with STDP, several features can be adjusted, such as training set size, timeouts, the number of neurons in each layer, the number of training episodes, and testing episodes. This allows for easy customization of the neural network's training and testing process while allowing users to analyze how each change affects the processes. You can also toggle

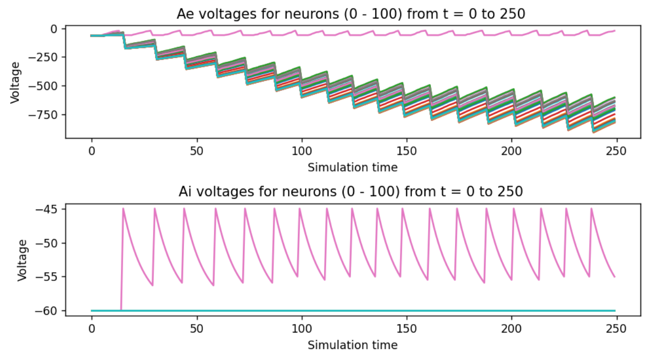


Fig. 1. Shows two line graphs labeled "Ae voltages for neurons (0 - 100) from t = 0 to 250" and "Ai voltages for neurons (0 - 100) from t = 0 to 250". During the simulation, these plots represent the membrane potential over time for two different types of neurons, possibly excitatory (Ae) and inhibitory (Ai). The Ae neurons show a step-wise decrease in membrane potential, while the Ai neurons exhibit oscillatory behavior, a periodic firing to a periodic input. [bin 2018]

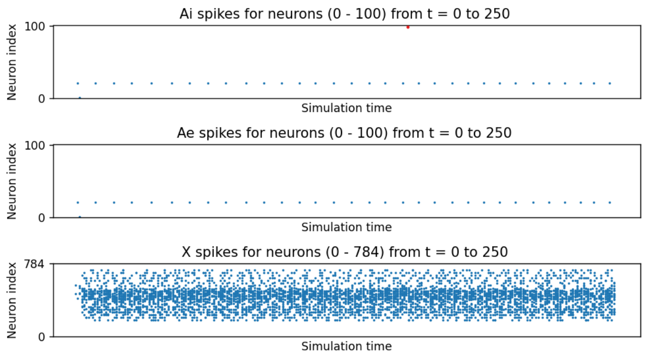


Fig. 2. Shows three raster plots with points scattered across the x-axis (Simulation time) and the y-axis (Neuron index). The plots are labeled "X spikes for neurons (0 - 784) from t = 0 to 250", "Ai spikes for neurons (0 - 100) from t = 0 to 250", and "Ae spikes for neurons (0 - 100) from t = 0 to 250". The "X spikes" plot has a much larger index range, corresponding to input neurons representing the MNIST images (since MNIST images are 28x28 pixels and 28 squared is 784). [bin 2018]

the plots on and off using command-line arguments. Examples of such plots are shown in Figure 3.[gen 2019][uns 2015]

These graphs are updated every time a new image is inputted into the network, and they are akin to how each new input causes the neurons to change and adjust their weights. This allows the network to learn the distinct features of each and to apply that in testing. So now the question becomes, how long does the training process take? And how accurate is it?

3.2 CPU vs GPU: BindsNET

While it may appear at first sight that the GPU would be significantly better than the CPU at training and testing a Spiking Neural Network, that was not the case. The CPU on a local laptop (Intel

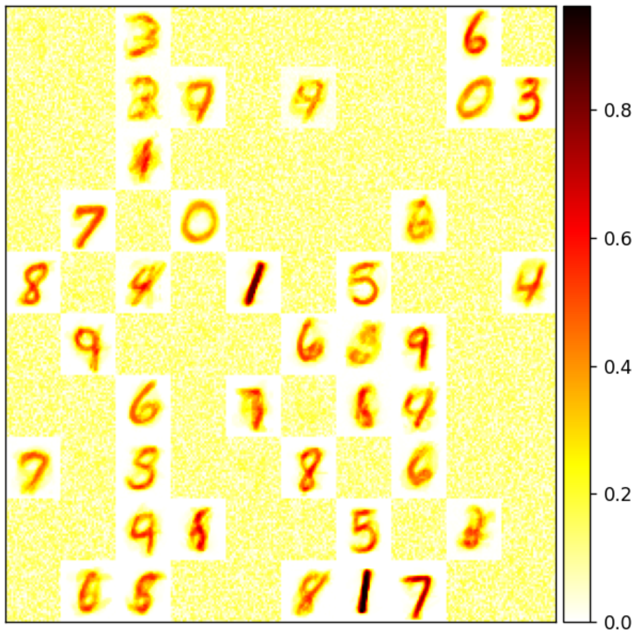


Fig. 3. Shows a heatmap of MNIST digits. The neurons communicate with discrete spikes, and these interactions are visualized with heatmaps to show the pattern of connectivity or activity. Since the model is trained to recognize handwritten digits, the heatmap shows areas where neuron activity is high for particular digits. The varying intensity suggests different response levels, which could correlate with how well the network has learned to recognize each digit. [bin 2018]

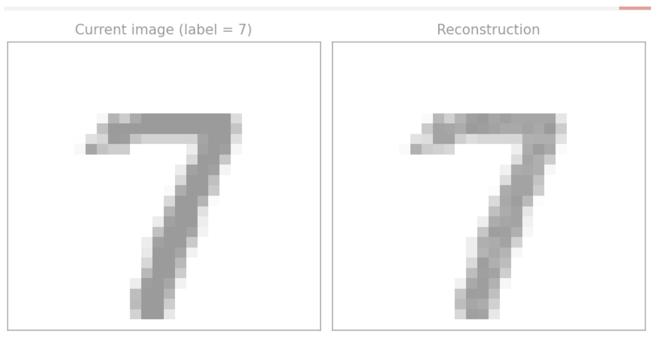


Fig. 4. Shows a comparison between the image currently being inputted into the network and the "reconstruction" output after processing the image. While the network has successfully reconstructed the number "7", some details are lost. [bin 2018]

i5 12500H) took approximately 23000 seconds to fully train the SNN on the MNIST dataset without batching the images, while the GPU (Nvidia 3050ti) took more than 3.0 times the CPU in training. The GPU was also training much slower than the CPU, which was 1.16 iterations per second, whereas the CPU was 2.61 iterations per second. This was the average of several trials running the dataset on the CPU and GPU. However, the gap suddenly diminishes when

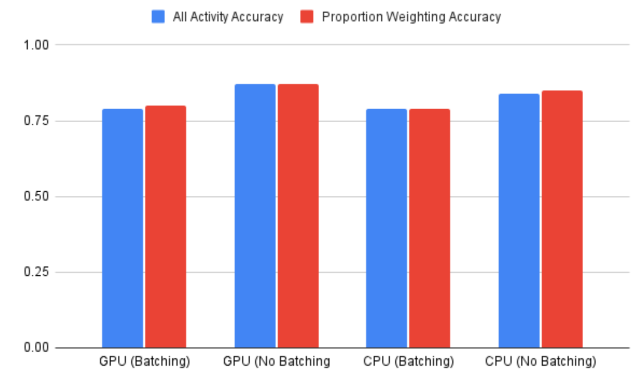


Fig. 5. Shows the training accuracy BindsNET achieved when working with BindsNET

batching the dataset and putting multiple images into the network as a single image. The CPU and the GPU finished training at 5621.6795 seconds at 10.61 iterations per second. While there were evident discrepancies between the CPU and the GPU, the more important question becomes which was more accurate. [und 2020]

Without batching, the GPU was slightly more accurate at 0.87 and the CPU at 0.84 regarding All Activity Accuracy. However, when using batching, the GPU and the CPU were the same, at 0.79 in terms of All Activity Accuracy. While it makes sense that the GPU, which took longer, has higher accuracy when not batching, why is the accuracy when batching the same as the CPU?

The difference between the GPU and CPU can likely be attributed to the differences in architecture and how each component was installed on the local laptop. In terms of architecture, the GPU benefits more from the batching process as it allows the GPU to utilize not only more of its processing capacity but also its parallel processing capabilities (though SNN, which utilizes event-driven mechanisms, may not fully utilize this). This is likely why the GPU could match the CPU in training speeds when using batching and not when batching was not used. This also explains why there is some accuracy loss when using batching. When the MNIST images are batched together, it introduces noise into the training process, which can act as a form of regularization, potentially leading to better generalization and hence similar accuracy across both devices and lower accuracy compared with not batching. GPUs also often use mixed precision to speed up computation, which can be another reason for the effect on accuracy. It's also possible that the specific SNN implementation in BindsNET itself is not optimized for GPU acceleration, leading to the underutilization of GPU resources. Memory transfer overheads between CPU and GPU can also contribute to slower performance when not using batching. With batching, memory transfers are more efficient as more significant data blocks are moved less frequently, reducing the relative overhead and allowing the GPU to match the CPU's performance.

The differences in training time and network accuracy show the importance of developing SNN software alongside the hardware. If the hardware cannot fully keep up with the computations, an SNN wouldn't be as accurate and fast. However, the goal of BindsNET was never to create excellent neural networks that could handle many tasks. It was designed for fast, quick, and straightforward prototyping specifically geared toward machine learning and reinforcement learning.

3.3 The Goal of BindsNET

The goal of BindsNET aligns with the broader trend in computational neuroscience and machine learning: to create tools that facilitate rapid prototyping and iterative design. This focus on agility and adaptability allows researchers and developers to explore the vast landscape of neural network architectures with relative ease despite the complex structures of neural networks. In serving its purpose, BindsNET lowers the barrier to entry, enabling a broader audience to experiment with SNNs without the steep learning curve typically associated with this field. The insights gained from such explorations are not merely academic; they have practical implications for designing neuromorphic hardware and developing algorithms that could one day enable more efficient, brain-like computation. Furthermore, by providing a framework where the complexities of SNNs are abstracted away, BindsNET allows users to focus on their models' conceptual and theoretical aspects rather than the technical and physical components. This is essential for advancing our understanding of how spiking networks can be harnessed for tasks such as pattern recognition, decision-making, and sensory processing.

In the broader context of AI development, the trade-offs observed between speed and accuracy in different hardware configurations underscore the need for co-optimizing algorithms and hardware. As neuromorphic engineering progresses, the co-evolution of software like BindsNET with specialized neuromorphic processors may pave the way for the next generation of AI systems that are powerful, energy-efficient, and capable of real-time performance. In summary, BindsNET is an essential tool in neuromorphic computing. Its design philosophy emphasizes the importance of accessibility and rapid experimentation, which are critical to the iterative discovery process that drives both artificial intelligence and our understanding of the brain. As software and hardware for SNNs evolve, tools like BindsNET will play a pivotal role in bridging the gap between theoretical insights and practical applications.

4 SNN SIMULATORS: NENGO

In contrast to the BindsNET interface, Nengo uses primarily a web browser GUI to run the simulation. The installation process is straightforward, without creating a virtual environment. Installing the software starts only with pip install Nengo. The process of installing and developing looks like this:

1)Run pip and install Nengo in the terminal or the Windows Powershell.

2)Executing Nengo with Python Nengo in the terminal or the Windows Powershell.

3)Creating a new model: Within the Nengo GUI, you can start a new model by clicking on the "File" menu and selecting "New." This opens a new tab with a blank workspace where you can begin designing your model.

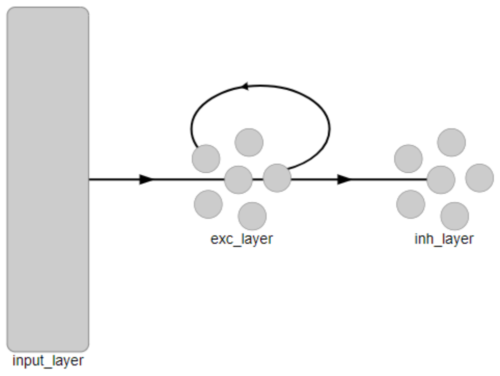


Fig. 6. A network in Nengo created to analyze MNIST

4)Building your model: Use the Nengo GUI to add neurons, connections, and input and output elements to your model. You can drag and drop elements from the toolbar and configure their properties through the GUI interface.

5)Writing or editing the model script: While Nengo GUI allows for visual programming, you can also directly edit the Python script that defines your model. This script is visible in the editor panel of the GUI, and you can modify it to fine-tune the behavior of your neural network.

6)Simulating the model: Once your model is ready, you can simulate it by clicking the "Start" button in the Nengo GUI. This will execute the model, and you can observe the results in real-time. The GUI also provides visualizations such as graphs and spike plots to help you analyze the neural activity.

The simplicity of starting with Nengo helps you get started quickly, and Nengo also comes with its learning tools in several tutorials for different SNN-based mechanisms. These start with a basic setup of an input, the exhibition neuron, and then the output. The input could range from many different values depending on the dataset, and the output may be the accuracy of the trained network. Some of these tutorials include: the 2D Decision Integrator, the Controlled Oscillator, and more. To compare with BindsNET, the code segment used in Nengo works with the MNIST dataset imported from the tensor library. MNIST was not a default dataset used in the examples given by Nengo. This is likely because Nengo intends for the software to focus more on simulating SNN in regards to brain structure. There isn't a training process; instead, the simulator visualizes how each neuron interacts with each other. In Figure 4, the images show the components that appear in a simple network.

Looking at this figure, let's analyze the input_layer. This is the first layer of the network, where input signals or data are received. In a biological context, this could represent sensory input; in a machine-learning context, it might be the features of a dataset. Next is the exc-layer, which stands for "excitatory layer." This middle group of neurons receives input from the input layer. Excitatory neurons increase the activity of the neurons to which they connect. In this diagram, the exc-layer is shown to have recurrent connections (the loop), indicating that neurons in this layer also send signals to each other. This can allow for the creation of complex dynamics, such as oscillations or persistent activity, which are essential for processes like memory or computational operations in the brain or an artificial neural network. And finally, the Inh-layer. This is the "inhibitory layer," composed of neurons that inhibit the activity of other neurons. These neurons receive input from the exc-layer, and their function is typically to regulate or suppress the activity within a network, preventing runaway excitation and ensuring stable network dynamics. Inhibitory connections are essential for neural networks to function correctly, as they control the overall level of neural activity, shape the timing of outputs, and contribute to learning mechanisms by balancing excitation with inhibition. The arrows indicate the direction of signal flow or information processing. Starting from the input-layer, the signal flows to the exc-layer and then from the exc-layer to both itself (recurrently) and the inh-layer.

4.1 Nengo MNIST

While the simulation of MNIST was possible through Nengo, there isn't much data to speak of. While BindsNET allowed the testing of SNNs to obtain accuracy, Nengo did not. Instead, it shows the process of inputting values into the neurons and outputting them in graphs and tables.

Figures 8 to 12 are plots given by inputting the MNIST dataset into Nengo.

4.2 The Goals of Nengo

Nengo was created with goals different from BindsNET. While BindsNET focuses on fast prototyping and setting neural networks, Nengo is a software tool used to build and simulate large-scale models based on the Neural Engineering Framework (NEF) and teach how the NEF is used. Its simulating capabilities allow students and researchers to explore how various cognitive functions can be implemented in neural circuits, providing a practical and theoretical understanding of brain function while not requiring specific hardware to fully utilize and learn about neural functions. It is also made for doing research that generates specific NEF models to explain experimental data. Researchers use it to develop and test hypotheses on the neural underpinnings of cognition, sensory processing, and motor control. By simulating neural models ranging from simple sensory systems to complex cognitive tasks, Nengo helps explore how the brain processes information and executes functions. Nengo integrates seamlessly with various scientific computing libraries, enhancing its utility for complex computational tasks. This integration allows researchers to combine nengo's modeling capabilities with robust data analysis and processing tools, broadening the scope of its applications in neuroscience and cognitive modeling.[nen 2013][nen Year][nen 2018][nen 2009]

In neuromorphic computing, Nengo allows users to visualize how SNNs are modeled after brain functions through graphs and network components. Nengo capitalizes on the importance of reaching a wider audience instead of specialized experts, allowing beginners to understand SNNs while ensuring that the application is well-developed for experts. This dual focus will enable Nengo to support various projects, from educational simulations in classroom settings

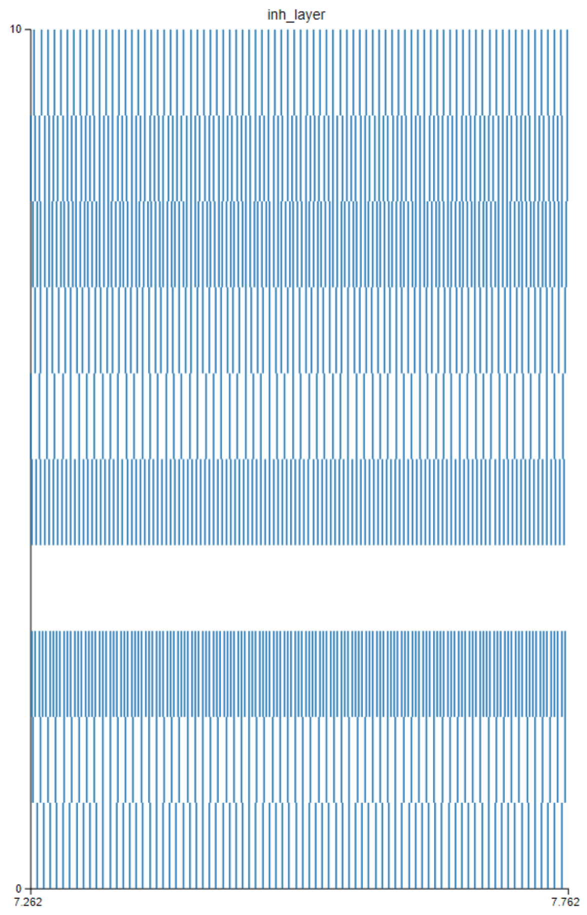


Fig. 7. A graph showing the inputs' spikes using horizontal lines in the inhibitory layer. The x-axis shows the time in seconds while the y-axis shows the strength of the spike, giving it a range from 0 to 10

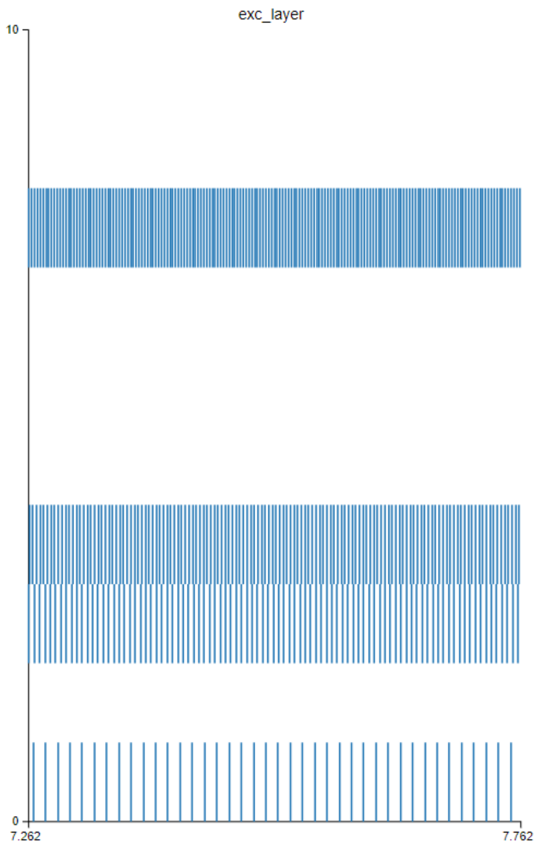


Fig. 8. Similar to the inhibitory layer graph, this graph shows the spikes of the excitatory layer during the same time range using the same x-axis and y-axis of seconds and strength

to cutting-edge research in large-scale neuromorphic systems. Overall, Nengo serves as a bridge between theoretical neural engineering and practical application, providing tools as educational as they are powerful. Its ability to cater to newcomers and experts makes it a pivotal platform in the evolving landscape of neuromorphic computing. It drives forward innovations that could become integral to how computers are designed and function.

5 COMPARING NENGO AND BINDSNET

While Nengo and BindsNET are SNN simulators, their philosophies differ. Nengo seeks to bridge the gap between theoretical neural engineering and practical application, while BindsNET is more of a tool for rapid prototyping and setup.

5.1 Nengo

Pros:

Biological Plausibility: Nengo excels in simulating neural models that closely mimic biological processes, making it invaluable for research that aims to understand or replicate brain functions in computational models.

Built for NEF: Nengo is built for large-scale models based on the NEF and allows users to simulate large models quickly and to collect large amounts of data for subsequent analysis

Educational Value: Its comprehensive approach to modeling makes it an excellent tool for education, helping students and researchers grasp complex neural dynamics and their applications.

Versatile Application: Nengo supports the development of practical applications such as robotics, cognitive models, and real-time simulation systems, demonstrating its versatility beyond academic research.

Graphical User Interface (GUI): Nengo provides a user-friendly browser-based GUI for designing and simulating networks, allowing users to interact with the models and components.

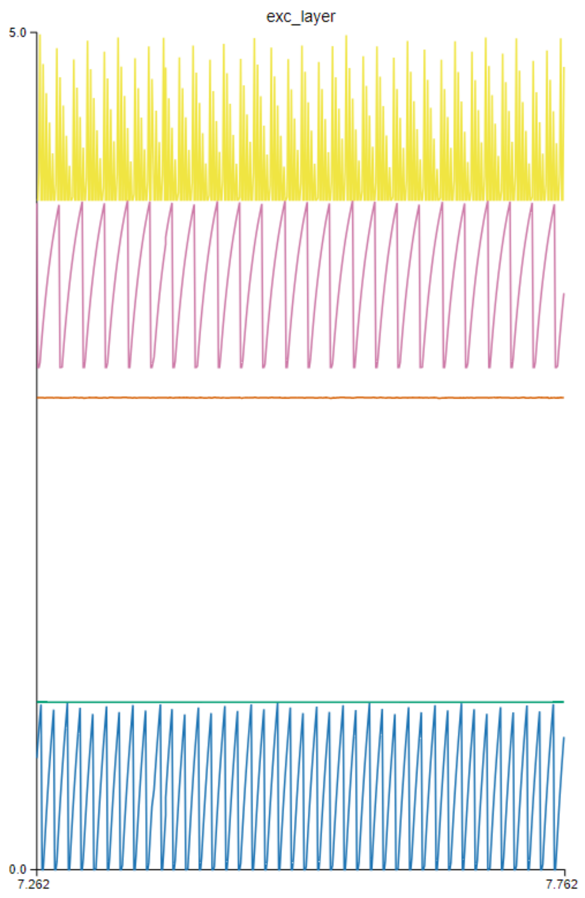


Fig. 9. The yellow lines at the top indicate a group of neurons with high-frequency firing, potentially in response to a strong or specific input. The purple lines show another group with a different, possibly slower firing pattern, which might indicate different types or classes of excitatory neurons within the layer or neurons responding to a different aspect of the input. The blue lines at the bottom suggest a baseline activity or a group of neurons with minimal response, possibly inhibited or not directly stimulated during this interval.

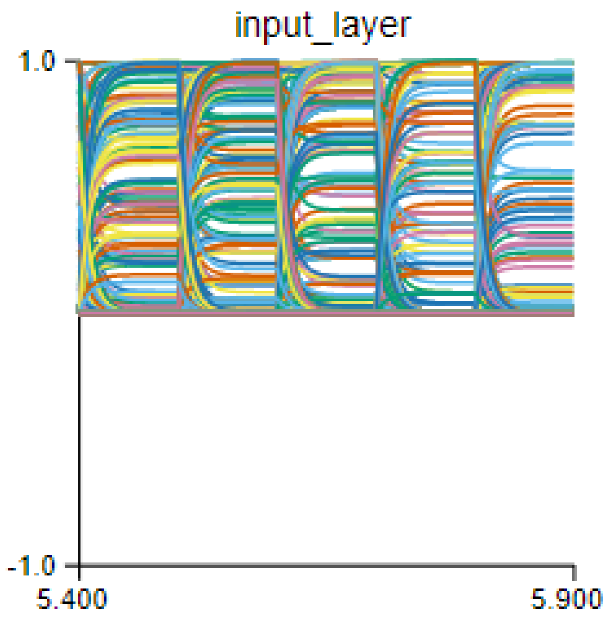


Fig. 10. This plot displays the inputs to a neural network layer over time, where each color represents a different input feature or channel. The wavy lines indicate continuous changes in input values, representing the varying inputs of MNIST. The x-axis is time in seconds, and the y-axis shows the amplitude

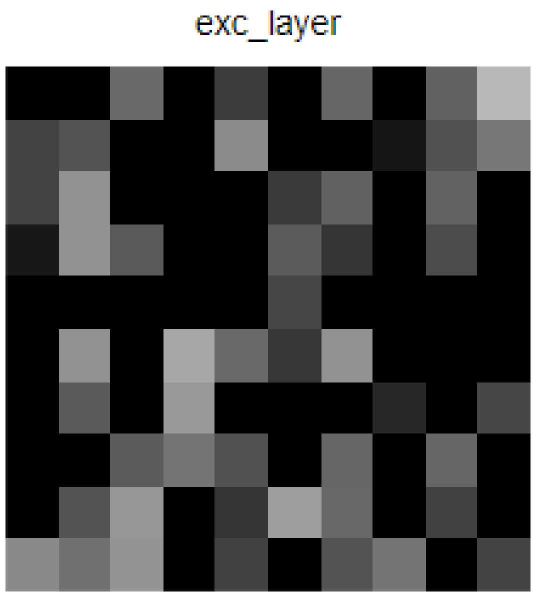
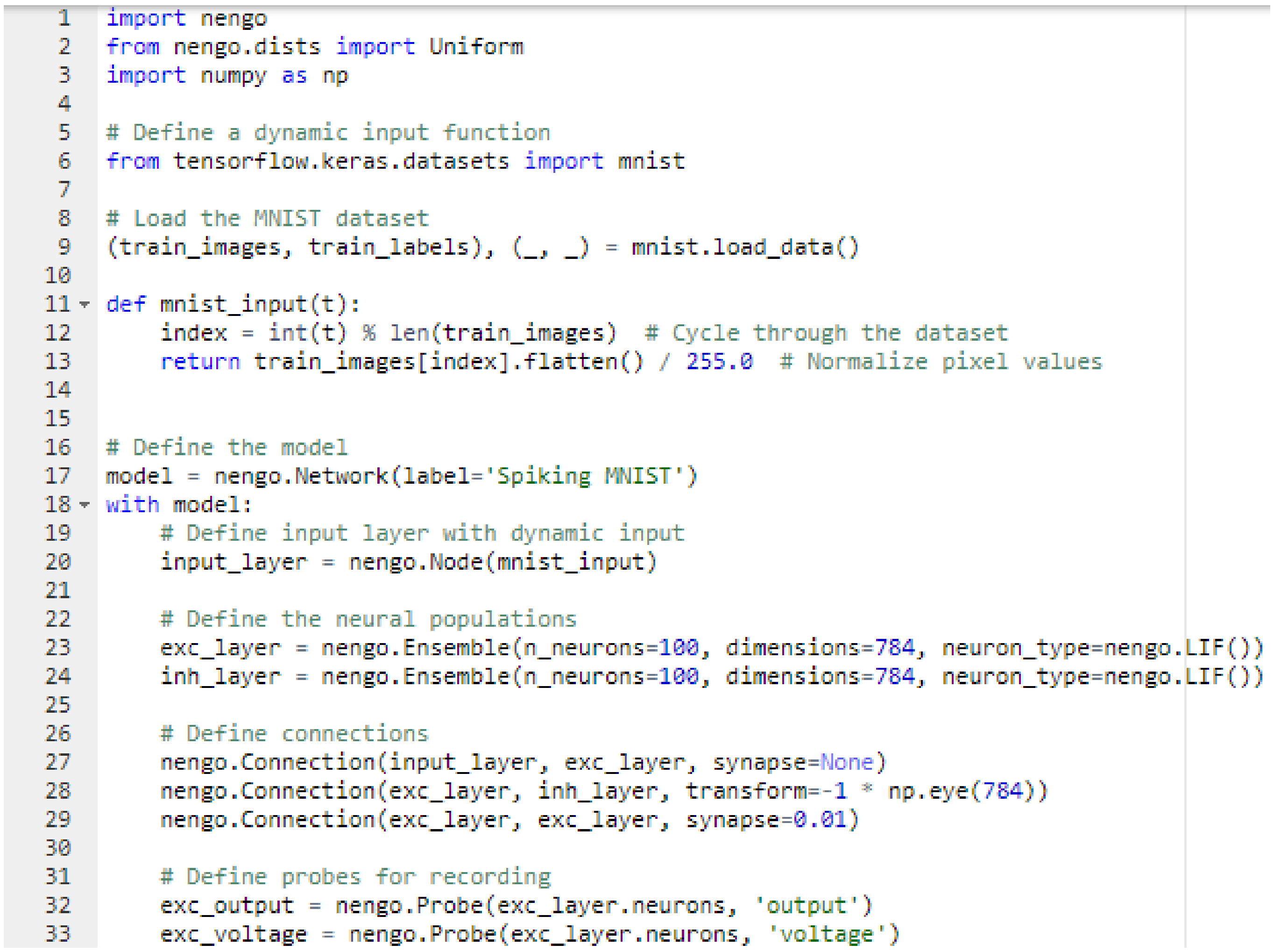


Fig. 11. A raster plot for the excitatory layer

Ease of Use: Nengo is extremely easy and straightforward to understand. The process of starting nengo starts with just "python nengo," and then the first model that appears on the browser is an example of how a possible network might appear. [mac [n. d.]][nen 2018][pyt 2019][spi 2020][und 2020][opt 2020][spi 2020] Cons:

Computational Demand: Simulating large, detailed neural models can be computationally intensive, requiring significant resources, which might limit its use in resource-constrained environments, especially since users cannot specify the exact computation hardware to use without further implementations.



```python
import nengo
from nengo.dists import Uniform
import numpy as np

# Define a dynamic input function
from tensorflow.keras.datasets import mnist

# Load the MNIST dataset
(train_images, train_labels), (_, _) = mnist.load_data()

def mnist_input(t):
    index = int(t) % len(train_images)  # Cycle through the dataset
    return train_images[index].flatten() / 255.0  # Normalize pixel values


# Define the model
model = nengo.Network(label='Spiking MNIST')
with model:
    # Define input layer with dynamic input
    input_layer = nengo.Node(mnist_input)

    # Define the neural populations
    exc_layer = nengo.Ensemble(n_neurons=100, dimensions=784, neuron_type=nengo.LIF())
    inh_layer = nengo.Ensemble(n_neurons=100, dimensions=784, neuron_type=nengo.LIF())

    # Define connections
    nengo.Connection(input_layer, exc_layer, synapse=None)
    nengo.Connection(exc_layer, inh_layer, transform=-1 * np.eye(784))
    nengo.Connection(exc_layer, exc_layer, synapse=0.01)

    # Define probes for recording
    exc_output = nengo.Probe(exc_layer.neurons, 'output')
    exc_voltage = nengo.Probe(exc_layer.neurons, 'voltage')
```

Fig. 12. The code segment used in Nengo for simulating MNIST

Less Focus on Rapid Prototyping: Compared to BindsNET, Nengo may not be as efficient for scenarios where rapid testing and iteration of network models are required.

5.2 BindsNET

Pros:

Rapid Prototyping: BindsNET allows users to easily and quickly configure the project through specific parameters.

Integration with PyTorch: BindsNET is built on top of the PyTorch library, allowing users to switch between GPU and CPU.

Flexibility: It allows for exploring and implementing various learning rules and network architectures without deep dives into neural dynamics, making it suitable for explorative research and application development.

Resource Efficiency: BindsNET is optimized for performance, making it less demanding on computational resources than more detailed simulators.

Cons:

Less Emphasis on Biological Details: While it supports spiking neural networks, BindsNET does not offer the same level of biological fidelity as Nengo, which is a drawback for users focused on neuroscientific accuracy.

Lack of Bridge From Theoretical to Application: Users might find it challenging to apply findings from BindsNET simulations to theoretical neuroscience or to validate models against biological evidence.

Terminal Interface: Without an interactive interface, it's harder for users without an extensive programming background to understand exactly what is happening.

5.3 Summary

Choosing between Nengo and BindsNET largely depends on the specific needs and goals of the project. If the priority is to achieve high biological fidelity and integrate with robust theoretical frameworks for deep scientific inquiries, Nengo is preferable. Conversely, if the focus is on quick development cycles, flexibility in network design, and integration with modern machine learning workflows, BindsNET would be more suitable. Overall, Nengo is designed to be better suited for beginners as it has a more straightforward interface and is a quick start into neural networks compared with BindsNet, which can be somewhat user-specific. However, each tool contributes to the neuromorphic computing community and, when used together, allows for a comprehensive learning process of SNNs.

6 DISCUSSION/DIFFICULTIES

While this paper primarily focused on comparing Nengo and BindsNET, other simulators have been developed atop Nengo or BindsNET or are independent of the two. Some of these examples were briefly mentioned in Section 2, such as GenericSNN, SuperNeuro, and EvtSNN.

6.1 GenericSNN: A Framework for Easy Development of Spiking Neural Networks (2024)

GenericSNN is an SNN framework intended to make SNN experiments accessible to researchers without deep programming skills. It offers a modular design where users can customize neuron models, learning rules, and network architectures through simple configuration files. The framework supports various SNN models and learning mechanisms, and it can run on both CPU and GPU, making it versatile for different computational environments. [Martin-Martin et al. 2024]

6.2 SuperNeuro: A Fast and Scalable Simulator for Neuromorphic Computing (2023)

SuperNeuro is a high-performance, scalable simulator that addresses the limitations of current simulators that are either too slow or not scalable when dealing with large networks and do not offer all the features needed for neuroscience applications such as BindsNET. This Python-based simulator supports CPU and GPU executions and is compatible with different simulation strategies, including matrix computation (MAT) and agent-based modeling (ABM). Based on the results collected by SuperNeuro, it exceeds BindsNET in terms of computation speed, particularly evident in simulations of small networks where SuperNeuro is 61x as fast and for larger sparse networks, a speedup of more than 3x. This simulator also facilitates rapid testing and development of neuromorphic algorithms, potentially enhancing the design of neuromorphic hardware.[sup 2020]

6.3 EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering (2022)

EvtSNN realizes some of the issues with a clock-driven method used by other simulators such as BindsNET and creates an event-driven simulator written in C++ that uses pre-filtering to avoid unnecessary computations and clusters neurons into populations. This simulator continues a previous event-driven simulation framework, the EDHA (Event-Driven High Accuracy). The framework's core task was to maintain pulse priority queue, however, they found computation limitations with this approach. Using pre-filtering and clustering neurons, they achieved an 11.4-time speedup compared to EDHA. (This paper also analyzed BindsNET alongside other simulators, showing that using GPU doesn't necessarily improve training speed)[evt 2022]

6.4 Difficulties with LAVA

Another SNN framework/simulator attempted was LAVA, an open-source software library dedicated to developing algorithms for neuromorphic computation. Lava allows users to run and test neuromorphic algorithms on traditional von Neuman CPUs. Still, more importantly, it will enable neuromorphic algorithms to run on research chips like the Intel Loihi chips. However, the installation process was extraneous. The installation method required the Python poetry library; however, when attempting to run poetry to configure the virtual environment, poetry did not exist despite already being installed using pip. This led to reinstalling poetry many times until it worked for one time only. Then, further problems arose during that trial as all the tests that ran failed because of the extensive packages that LAVA utilizes. This led to many pip installs and uninstalls that later proved ineffective. The difficulties in installing LAVA were one of the main reasons for switching the comparison between BindsNET and Nengo rather than BindsNET and LAVA. Other reasons included the differences in purpose and the difficulties in understanding by non-experts. While Nengo and BindsNET were simple and easy to understand and use, LAVA did not have the same goals.

6.5 Future Work

Working with BindsNET and Nengo provides a foundation for understanding what is needed for a successful SNN simulators and in the future, working with more modern simulators like GenericsSNN or SuperNeuro or EvtSNN can provide a more current view into the state of SNN simulators. This would also be a great segway into creating a more complete repository of SNN simulators comparing all the benefits, but also the downsides. With those simulators, it would also be crucial to work with datasets other than MNIST that are more suited for SNNs such as Coco or ImageNet which include more natural images other than just numbers. This would allow the network to better utilize its temporal aspect and its capacity for image classification as these datasets are much more diverse.

7 CONCLUSION

The discrepancies that Nengo and BindsNET fill in developing and understanding Spiking Neural Networks are unique and allow users to learn how SNNs are built. Both are easy to install and are Python-based, naturally offering adequate abstractions. Nengo has a neater interface based on GUI, while BindsNET is based on the terminal, which is not as tidy. However, though BindsNET is designed mainly for rapid prototyping rather than neuroscience-based applications, an interface that is easy on the eyes can be crucial to understanding the training process. One significant issue with both simulators, though, is the lack of measurement in terms of energy efficiency. As carbon emissions continue to be a substantial problem, we should look for ways to reduce as much as possible. While the thinking that SNNs are indeed energy efficient, it should be an essential part when identifying issues to creating better SNNs. This would also allow researchers to understand their prototypes' costs when put in reality.

ACKNOWLEDGMENTS

Thank you to Professor Lillian Pentecost for guiding me on fixing problems encountered when using some of the simulators.

REFERENCES

[n. d.]. ([n. d.]).

2009. Python scripting in the Nengo simulator. https://www.frontiersin.org/articles/10.3389/neuro.11.007.2009/full

2013. Nengo: a Python tool for building large-scale functional brain models. https://www.frontiersin.org/articles/10.3389/fninf.2013.00048/full

2015. AXNN: energy-efficient neuromorphic systems using approximate computing. *ACM Digital Library* (2015). https://dl.acm.org/doi/abs/10.1145/2627369.2627613

2015. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Neuroscience* 9 (2015), 376. https://www.frontiersin.org/articles/10.3389/fncom.2015.00099/full

2018. BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python. https://www.frontiersin.org/articles/10.3389/fninf.2018.00089/full

2018. Nengo and Low-Power AI Hardware for Robust, Embedded Neuromobotics. https://www.frontiersin.org/articles/10.3389/fnbot.2018.00068/full

2019. Benchmarking Spiking Neural Networks with Multiprocessing. https://oshears.github.io/assets/docs/vet_ece_5510_project_report.pdf

2019. GeneVision: A Framework for Easy Development of Spiking Neural Networks. https://ieeexplore.ieee.org/abstract/document/10506505

2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288efe2392f2bfa0f127740-Paper.pdf

2019. Spiking neural network decoder for brain-machine interfaces. https://ieeexplore.ieee.org/abstract/document/5910570

2020. Deep Spiking Neural Network: Energy Efficiency Through Time Based Coding. file:///C:/Users/alsto/Downloads/978-0-303-58607-2_23.pdf

2020. Optimizing the Energy Consumption of Spiking Neural Networks for Neuromorphic Applications. *IEEE Transactions on Neural Networks and Learning Systems* 31, 4 (2020), 1205–1217. https://ieeexplore.ieee.org/abstract/document/9035426

2020. Spiking Neural Networks Hardware Implementations and Challenges. https://dl.acm.org/doi/abs/10.1145/2627369.2627613

2020. SuperNeuro: A Fast and Scalable Simulator for Neuromorphic Computing. https://dl.acm.org/doi/abs/10.1145/3383977.3360060

2020. Understanding SNN and Its Recent Advancements. https://engrxiv.org/preprint/view/3652

2021. Sparse Distributed Memory using Spiking Neural Networks on Nengo. https://arxiv.org/abs/2109.03111

2022. EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering. https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.944262/full

Unknown Year. NengoFPGA: an FPGA Backend for the Nengo Neural Simulator. https://uspace.waterloo.ca/handle/10012/14923

Alberto Martin-Martin, Marta Verona-Almeida, Rubén Padial-Allué, Javier Mendez, Encarnación Castillo, and Luis Parrilla. 2024. GenericSNN: A Framework for Easy Development of Spiking Neural Networks. *IEEE Access* 12 (2024), 57504–57518. https://doi.org/10.1109/ACCESS.2024.3391889

Received 12 May 2024; accepted 12 May 2024