Comparative Study of Open-Source Large-Scale SNN Simulators for Pattern Recognition and Image Classification

Abhinav Barman
Department of Electrical Engineering
Indian Institute of Technology Ropar

Gaurav Wani
Department of Electrical Engineering
Indian Institute of Technology Ropar

Ranjeet Singh
Department of Electrical Engineering
Indian Institute of Technology Ropar

***Abstract*—This report presents a comparative study of open-source large-scale Spiking Neural Network (SNN) simulators for pattern recognition and image classification. Various simulators are evaluated based on accuracy, computational resources, and compatibility with existing Convolutional Neural Networks (CNNs). This study aims to provide insights into the performance and usability of SNN simulators for real-world image recognition applications.**

***Index Terms*—Spiking Neural Networks, Convolutional Neural Networks, Pattern Recognition, Image Classification, SNN simulators, and Backpropagation.**

I. Introduction

With the increasing data processing demand in edge devices, these devices must be made capable of processing data themselves rather than sending them to the server. This saves time, but with the increasing amount of data being received by these devices due to recent advancements in the Internet of Things(IoT), there is a need for edge devices to process information in intelligent and useful ways, which triggers the need for neuromorphic architectures. These architectures are a big step towards Beyond the Moore Era instead of the binary system used in Von Neumann's architecture. Neuromorphic Architectures are inspired by the working of the brain which processes information by the spiking of the neurons. Since a large amount of data will be received on the edge devices, an efficient method to process the important data is required. This report suggests using Spiking Neural Networks to process the data and train it for image recognition. We also compare the results with Convolutional Neural Networks and observe the performance of both models on various parameters. Since the increasing complexity of the CNN models demands increased computational power, the power efficiency provided by the SNN models really helps us prevent this much usage of power.

II. Literature Review

The human brain exhibits remarkable computational efficiency, excelling at tasks like pattern recognition, decision-making, and working memory. This has inspired researchers to develop neuromorphic architectures. These architectures aim to mimic the computational style of the brain using artificial systems built with CMOS technology.

Neuromorphic architectures [2] hold significant promise for overcoming the limitations of traditional Von Neumann architectures, particularly in areas like edge computing, where low power consumption and real-time processing are critical. Neuromorphic architectures leverage CMOS technology to create circuits that emulate the behavior of biological neurons.

Spiking Neural Networks (SNNs) are a core component of neuromorphic architectures. They process information through the transmission of discrete spikes (electrical pulses) over time. This approach aligns closely with how biological neurons communicate. SNNs hold the potential to explain various brain functions by offering a more biologically plausible computational framework compared to traditional artificial neural networks.

Neuromorphic architectures offer significant advantages over traditional Von Neumann architectures [3]. Here's a breakdown of their key differences:

- **Processing Style:** Neuromorphic architectures operate parallelly, contrasting with the sequential processing nature of Von Neumann architectures.
- **Memory and Processing:** In neuromorphic architectures, memory and processing are collocated, whereas they are separated in Von Neumann architectures.
- **Programming:** Neuromorphic architectures utilize Spiking Neural Networks (SNNs) for programming, while Von Neumann architectures code on binary instructions.
- **Communication:** Neuromorphic architectures employ spikes for communication, mimicking the biological approach. In contrast, Von Neumann architectures use binary data for communication.
- **Timing:** Neuromorphic architectures function asynchronously, driven by events, whereas Von Neumann architectures operate synchronously, driven by a clock.

These characteristics of neuromorphic architectures pave the way for the development of low-power, high-performance computing systems capable of tackling complex tasks similar to the human brain. Their parallel processing, efficient memory usage, and natural communication mechanisms inspired by the brain hold significant potential for future advancements in computing.

The ever-increasing amount of data generated at the network's periphery or edge is driving a shift towards **edge computing** [1]. Unlike traditional cloud-centric architectures, edge computing distributes processing tasks to devices located

closer to data sources, such as sensors. This decentralization offers several benefits:

- Reduced Latency: By processing data locally, edge computing minimizes the distance information needs to travel, resulting in faster response times and improved real-time performance for applications like autonomous vehicles or industrial automation.
- Enhanced Bandwidth Efficiency: By processing data on edge devices, we avoid bogging down the network. This is especially helpful for applications that use a lot of data, like high-definition security camera footage.
- Improved Reliability: Local processing offers greater autonomy and resilience in scenarios with limited or unreliable network connectivity. This is crucial for applications in remote locations or those requiring an uninterrupted operation.

With the increasing use of sensors in Internet of Things (IoT) applications, we need smart ways to handle all the data they collect at these resource-constrained edge devices. Traditional approaches to sending all data to the cloud for processing are no longer feasible due to bandwidth limitations and latency constraints.

Emerging hardware and AI for edge processing to address the growing data volume and processing needs at the edge, advancements are being made in both hardware and software domains:

- AI Chips: New chips are being designed specifically for edge devices. This allows these devices to run machine learning and neural networks directly without needing a computer. These chips often integrate functionalities like analog-to-digital conversion, filtering, etc.
- Emerging Devices: As traditional device scaling approaches its physical limits, alternative technologies like memristors are gaining traction. These devices offer the potential for increased processing speed and on-chip area due to their unique properties, including the ability to perform analog computations and map neural network architectures efficiently.

The limitations of traditional device scaling techniques are pushing the industry to explore new avenues beyond Moore's Law and conventional CMOS technology. This shift focuses on developing neural computing solutions that offer increased processing power and functionality at lower costs and energy consumption, a critical consideration for battery-powered edge devices.

Artificial Neural Networks (ANNs) have transformed image recognition, excelling in tasks like object detection and classification. Traditional architectures, notably Convolutional Neural Networks (CNNs), have achieved remarkable accuracy but often demand substantial computational resources, posing challenges for edge devices. These devices, crucial for real-time image recognition, require efficient solutions.

Spiking Neural Networks (SNNs) present a promising alternative for edge-based image recognition. Unlike CNNs, which rely on continuous activations, SNNs use spiking neurons and discrete electrical pulses (spikes). This event-driven nature inherently conserves energy, crucial for resource-constrained edge devices. SNNs also hold the potential for hardware efficiency, making them suitable for deployment. However, SNNs pose unique challenges. Training for image recognition differs from CNNs, as backpropagation relies on activation function differentiability. Techniques like surrogate gradients are necessary for effective training. Additionally, datasets optimized for SNNs are scarce, hindering their development.

Despite challenges, SNNs promise to revolutionize edge-based image recognition. Their energy efficiency and hardware potential make them ideal for applications like smart cameras and medical devices. Ongoing research aims to enhance SNN performance, positioning them as viable alternatives to CNNs for edge-based tasks.

The **Leaky Integrate-and-Fire** (LIF) neuron model is a fundamental concept in neuroscience and artificial intelligence [4]. It offers an effective representation of neuronal behavior, striking a critical balance between biological plausibility and practicality. Unlike more complex models like Hodgkin-Huxley neurons, which capture intricate biophysical details but are computationally expensive, the LIF model prioritizes efficiency.

In the LIF model, a neuron integrates incoming signals over time. If the integrated value surpasses a predefined threshold, the neuron emits a spike (action potential). This simplified approach focuses on the timing of spikes rather than detailed waveform characteristics. This focus on spike timing allows for efficient information encoding and processing, making the LIF model suitable for various computational tasks. The LIF model effectively bridges the gap between biological realism and computational efficiency in neural modeling.

Within a neural network, a single neuron can receive input signals from hundreds or even thousands of other neurons. These inputs can originate from various sources like sensory stimuli and other pre-synaptic neurons, which is the most common scenario. Due to the short duration of these input spikes and the inherent temporal dynamics of neuronal communication, it's highly improbable for all input spikes to arrive at the neuron body simultaneously. Instead, temporal dynamics sustains the influence of these inputs and delays its mechanism. The passive membrane surrounding a neuron plays a crucial role in its function. Composed of a lipid bilayer, this membrane acts as a capacitor, separating conductive saline solutions inside and outside the neuron. Specific channels embedded within the membrane control the movement of ions, regulating the neuron's electrical activity. This membrane behavior can be effectively modeled using electrical components like resistors and capacitors.

A crucial aspect of the LIF model is the reset mechanism. The sharp drop in membrane potential reduces the likelihood of subsequent spike generation, contributing to the brain's energy efficiency. The reset mechanism can be implemented in different ways:

- Reset by subtraction (default): Subtracting the threshold value from the membrane potential after every spike.

- Reset to zero: Forcing the membrane potential to zero upon spiking.
- No reset: Leaving the membrane potential unchanged, potentially leading to uncontrolled firing behavior.

Training Spiking Neural Networks (SNNs) presents unique challenges compared to traditional Artificial Neural Networks (ANNs). Unlike ANNs with continuous activation functions, SNNs utilize spiking neurons that communicate through discrete electrical pulses (spikes). This difference necessitates specialized training approaches to overcome hurdles like the "dead neuron problem." [4]

There are two primary approaches for training SNNs:

1) **Shadow Training:** This method leverages the well-established training algorithms for ANNs. Here, a traditional ANN is first trained on the desired task. Subsequently, the trained ANN is converted into an SNN by approximating the activation functions and learning the rules of the ANN using spiking neuron models. While convenient, shadow training may not guarantee optimal performance for SNNs, as the network architecture and dynamics might not perfectly translate between the two models.
2) **Backpropagation with Spikes:** This approach directly trains the SNN using a modified version of the backpropagation algorithm. Similar to ANN training, the goal is to minimize the error between the network's output (spike trains) and the desired output. The error is propagated backward through the network, adjusting the weights between neurons to progressively reduce the error.

The backpropagation process in SNNs typically involves these steps:

- **Initialization:** Weights in the network are randomly assigned.
- **Forward Pass:** The input data is propagated through the network, generating a sequence of spikes as the output.
- **Error Calculation:** A cost function measures the difference between the actual and desired output spike trains.
- **Weight Update:** This step employs the chain rule to calculate the gradient of the cost function with respect to the weights. The weights are then updated in a direction that minimizes the error.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial S}\frac{\partial S}{\partial U}\frac{\partial U}{\partial W} \tag{1}$$

However, a significant challenge arises during SNN training known as the "dead neuron problem." This phenomenon occurs when neurons become inactive for the entire training process, no spikes are fired. These inactive neurons are essentially insensitive to weight updates, therefore hindering the network's learning ability.

$$S(U) = \begin{cases} 0 & \text{if } U < U_{thr} \\ 1 & \text{if } U \geq U_{thr} \end{cases} \tag{2}$$

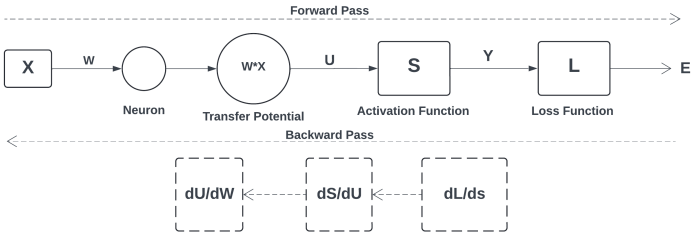

Fig. 1. Backpropagation to find error with respect to weight

$$\frac{dS}{dU} = \begin{cases} 0 & \text{if } U \neq U_{thr} \\ \infty & \text{if } U = U_{thr} \end{cases} \tag{3}$$

The standard backpropagation algorithm relies on the differentiability of the activation function used in the neurons. However, the spiking behavior of SNN neurons introduces non-differentiability issues, as seen above, by using the Heaviside Function as an Activation function for getting the output of neurons. During backpropagation, this leads to the formation of the Dirac Delta function, which is non-differentiable at threshold value and zero for other values. To address this, a technique called **Surrogate Gradient Descent** can be employed [4]. Here, the derivative of the activation function (e.g., the sigmoid function) used in the backpropagation algorithm is replaced with an alternative, differentiable function that approximates the behavior of the spiking neuron's membrane potential. This allows the network to continue learning even with inactive neurons present.

$$S(U) = \frac{1}{1 + e^{-U}} \tag{4}$$

One approach involves shifting and smoothing the sigmoid function to create a surrogate function. This function incorporates a parameter to control the smoothness of the approximation and enables effective training by overcoming the dead neuron problem. By utilizing surrogate gradients, SNN training can achieve effective learning despite the challenges posed by non-differentiability and inactive neurons. Therefore, after evaluating the loss function with respect to the weights using backpropagation, we apply gradient descent to update the weights.

$$W_{new} = W_{old} - \alpha\frac{\partial L}{\partial W} \tag{5}$$

III. Training SNNs using snnTorch

A. Python Library Adopted

snnTorch [8] is a lightweight and open-source Python library specifically designed for training and simulating Spiking Neural Networks (SNNs). It provides a user-friendly interface for its users to design, train, and evaluate SNN models. snnTorch offers efficient implementations of core SNN operations, such as functions for generating and manipulating

spikes, defining various spiking neuron models, and implementing different learning algorithms designed for SNNs. The library also integrates popular deep learning frameworks like PyTorch, allowing users to leverage existing tools and functionalities for building and training SNNs. This integration makes snnTorch a versatile and accessible platform for advancing research in SNNs and evaluating its performance metrics.

B. Dataset Description

N-MNIST is a unique dataset designed specifically for Spiking Neural Networks (SNNs). Unlike the classic MNIST dataset with static digit images, N-MNIST presents these digits as sequences of spikes. This spiking format captures the timing information of how a digit appears over time, mimicking how neurons communicate in the brain. This focus on temporal information is crucial for SNNs, as they rely on spike timing for processing. N-MNIST allows researchers to explore how SNNs handle visual tasks like digit recognition in a controlled environment, similar to traditional ANNs with MNIST. Additionally, it enables investigation into how SNNs leverage spike timing for information processing, a key advantage over ANNs.

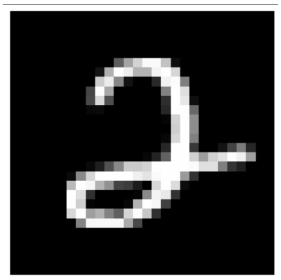

Fig. 2. MNIST DataSet Example

In the N-MNIST dataset, each row corresponds to a single event, which consists of four parameters: (x-coordinate, y-coordinate, timestamp, and polarity).

1. x and y coordinates correspond to an address in a grid.
2. The timestamp of the event is recorded in microseconds.
3. The polarity refers to whether an on-spike (+1) or an off-spike (-1) occurred, i.e., an increase or a decrease in brightness.

We can use other neuromorphic datasets like CIFAR10-DVS and DVSGestures for the same purpose.

C. Implementation Details

We use the Python **tonic** library to import and load the n-mnist dataset. Tonic streamlines the process of using the N-MNIST dataset for training Spiking Neural Networks (SNNs). Tonic simplifies data loading with user-friendly functions, similar to PyTorch for traditional datasets. Tonic is very good at handling this event-based format. It can directly feed the data into an SNN for training, and additionally, its users can leverage Tonic's built-in transformations. These transformations involve denoising the spike trains to remove unwanted noise, converting them into specific time frames compatible with the chosen SNN architecture, or even performing spatial or temporal manipulation to augment the dataset and improve the SNN's ability to generalize to unseen data. Hence, It simplifies data loading, handles the event-based format, offers optional transformations for data manipulation, and prepares the data for seamless integration with SNN training tools.

We employed an approach utilizing disk caching and batching to address the slow loading times associated with the original data format. This strategy consisted of writing loaded files from the dataset to disk for faster access. Keeping the variability in event recording lengths in mind, we used a custom collation function, 'tonic.collation.PadTensors()'. This function helped us in padding shorter recordings, ensuring consistent dimensions across all samples within a batch.

D. Defining the Neural Network

In the initial stages of the network, we used standard convolutional layers. The first convolutional layer extracts features from the input data using 12 filters with a kernel size of 5x5. This layer operates on data with two input channels, which represent grayscale images. Subsequent to feature extraction, a pooling layer performs downsampling to reduce the dimensionality of the data and introduce some level of translation invariance.

Following the standard convolutional layers, the network incorporates custom layers from the snnTorch library. These layers introduce spiking dynamics into the network, mimicking the firing behavior of biological neurons. The specific behavior is determined by parameters 'beta' (controlling leakiness of the activation function) and 'spike gradient' (influencing how gradients are calculated based on spiking behavior).

The architecture continues with another set of convolutional and pooling layers, extracting higher-level features and further reducing dimensionality. The data is then flattened to prepare it for a fully connected layer. The final fully-connected layer takes the flattened data and performs classification, with 10 output units corresponding to 10 digits. The final layer in the sequence is another SNN layer configured similarly to the previous ones.

Training Loop for Neural Network Optimization:

1. Initialization: The training loop begins by initializing two empty lists. One list will store the loss values calculated after each iteration, and the other will store the corresponding accuracy values. These lists will be used later to analyze the model's training progress.

2. Epoch Loop: The core training process is contained within an epoch loop. This loop iterates for a predetermined number of epochs, controlled by a hyperparameter specifying how often the entire training dataset will be passed through the network.

3. Mini-batch Loop: Nested within the epoch loop is a mini-batch loop. This loop iterates through the training data in smaller, manageable portions called mini-batches. For each mini-batch:

- **Data Loading:** Data and corresponding target labels are retrieved from the current mini-batch. These are then

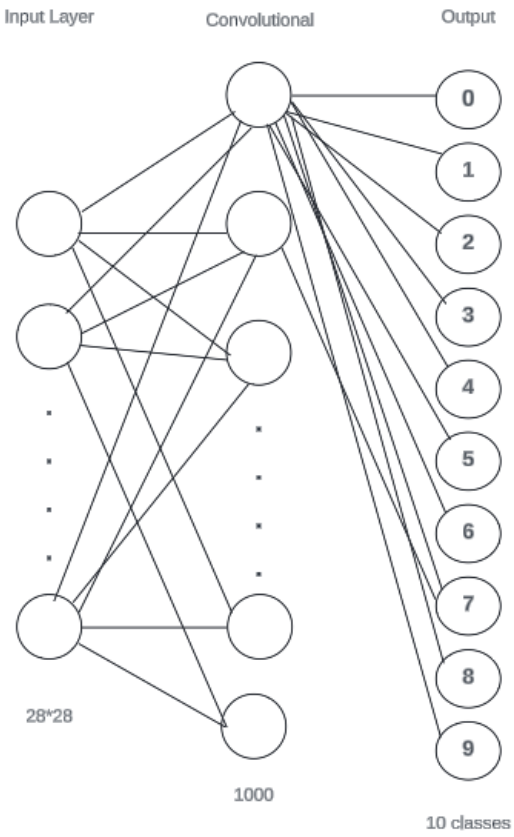

Fig. 3. Neural Network Model

transferred to the designated device (e.g., GPU) for faster computation during training.

- **Training Mode:** The network is switched to training mode. This enables functionalities specifically designed for the training phase, such as dropout or batch normalization, which can help improve the model's generalization capabilities.
- **Forward Pass:** The data from the mini-batch is passed through the neural network to obtain the model's output.
- **Loss Calculation**: A loss function is applied to evaluate the difference between the model's output and the true labels (ground truth) for the data in the mini-batch. This loss value represents the model's performance on the current mini-batch.
- **Backpropagation/Weight Update**: The calculated loss value is used to determine the gradients for the network's weights. These gradients indicate how adjustments to the weights can improve the model's performance in minimizing the loss. The network's weights are then updated based on these gradients using an optimization algorithm.
- **Performance Logging:** The loss value and accuracy for the current mini-batch are stored in the list for later training model analysis.

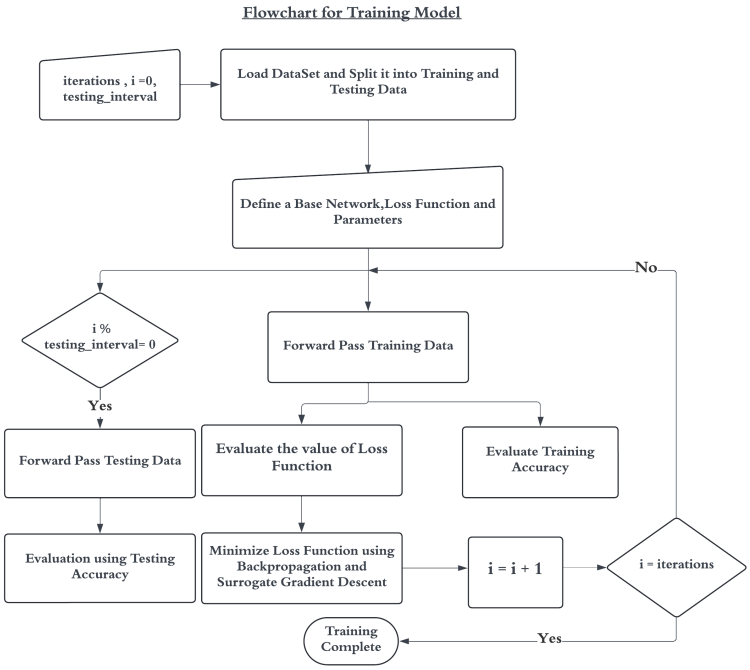

Fig. 4. Approach used to train SNN model

4. Loop Termination: The mini-batch loop completes after processing all mini-batches within the current epoch. The epoch loop then continues iterating through epochs until the pre-defined number of epochs is reached.

IV. Results

While training the neural networks using the architects of CNN and CSNN, we observed that CNNs are found to be more accurate, whereas when we checked the usage of memory while training, we found that CSNN used less memory in comparison to CNN. We can observe the accuracy of both models with their respective graphs.

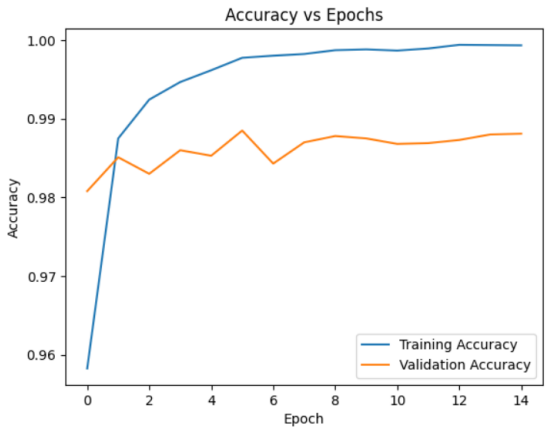

Fig. 5. Accuracy vs Epoch for CNN architecture

Fig. 6. Accuracy vs iterations for SNN architecture (200 iterations)

From training and evaluating the model, we observe that the accuracy in the case of SNNs is lesser than that of CNNs. This could be because, in SNN, we are training in batches for a given number of iterations. This leads to improper utilization of the entire data to train the SNN model. By increasing the number of iterations, we can use more images to train as well as evaluate the model. In [Fig. 7], we have trained the SNN model for 200 iterations. We can observe that the accuracy has increased from 89.8% to 93.5%.

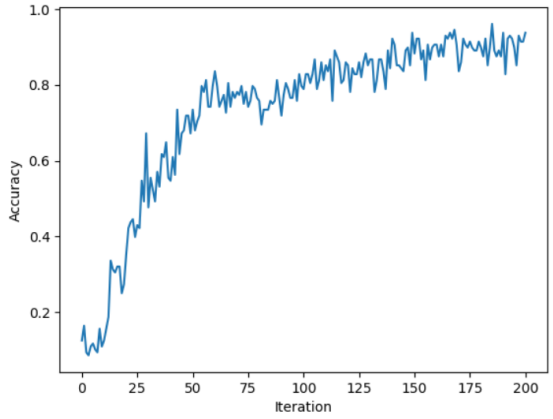

Fig. 7. Accuracy vs iterations for SNN architecture (100 iterations)

In CNN, we have trained using the entire dataset. We observed the testing accuracy to be 98.8% [Fig. 5]. Since neuromorphic datasets are larger in size as compared to static images, processing these files on our computers consumes more memory. Fortunately, by using neuromorphic architecture computing systems, the speed and power efficiency would be drastically lesser than traditional Von Neumann Architecture computing systems [1]. Since we were not using the respected compatible devices to evaluate models, we have not included the data for power consumption.

V. Comparison with other works on N-MNIST dataset

Sr. No	Model	Accuracy
1	STS-ResNet	99.6
2	FastSNN	95.91
3	CSNN	93.5
4	Sparse Spiking Gradient Descent	92.7

STS-ResNet [5], short for Deep Spatio-Temporal Residual Networks, is a sophisticated deep-learning model designed to predict crowd movement in urban environments. It operates on data that combines spatial information (location) with temporal information (timestamps), likely derived from historical records detailing the flow of people in different areas over time. The architecture of STS-ResNet incorporates residual networks, a type of neural network architecture optimized for capturing both long-term trends and short-term fluctuations in crowd dynamics. While STS-ResNet excels in analyzing complex spatiotemporal data, such as predicting future crowd inflow and outflow in various city regions, it may not be the optimal choice for the N-MNIST dataset. N-MNIST primarily consists of static images of handwritten digits, requiring simpler and more efficient models for pattern recognition tasks. Consequently, a standard Spiking Neural Network (SNN) is better suited for processing the N-MNIST dataset due to its effectiveness in recognizing patterns in static images.

Fast SNNs [6] are a specialized branch of research within Spiking Neural Networks (SNNs) that prioritize reducing inference speed. They achieve this objective through various techniques, such as pre-training Artificial Neural Networks (ANNs) for subsequent conversion to SNNs or employing specialized training algorithms. While Fast SNNs may incur a slight accuracy trade-off compared to meticulously trained regular SNNs, their primary value lies in real-time applications where rapid predictions are paramount.

Sparse Spiking Gradient Descent (SSGD) [7] is an innovative training algorithm tailored specifically for Spiking Neural Networks (SNNs). It addresses the inefficiencies inherent in applying traditional ANN training methods to SNNs. By capitalizing on the sparse nature of SNNs, wherein neurons are predominantly inactive, SSGD significantly reduces the computational burden of backpropagation during training. This approach yields comparable or superior accuracy to existing techniques while delivering substantial speed enhancements and improved memory efficiency. Consequently, SSGD emerges as a promising strategy for training SNNs, particularly when considering deployment on neuromorphic hardware platforms.

Convolutional Spiking Neural Networks (CSNNs) are a specialized type of Spiking Neural Network (SNN) designed for tasks involving spatial information like image and video recognition. They achieve this through convolutional layers similar to Convolutional Neural Networks (CNNs). Compared to general-purpose SNNs, CSNNs might be easier to train due to their CNN architecture. On the other hand, SNNs offer more general applicability for diverse tasks but can be more

challenging to train due to the discrete nature of spikes used in communication. CSNNs and SNNs share potential benefits like lower power consumption and a more biologically plausible approach to computation than traditional ANNs.

VI. Conclusion

Our comparative study of open-source large-scale SNN simulators reveals valuable insights for pattern recognition and image classification tasks. Evaluating simulators based on accuracy, computational resources, and ease of integration with CNN architectures highlights their strengths and limitations.

The study emphasizes the potential of SNNs for edge-based image recognition due to their energy efficiency and potential hardware advantages. However, challenges such as specialized training techniques and the limited availability of optimized datasets need to be addressed to fully leverage SNN capabilities.

In the future, researchers and developers should prioritize the development of efficient training algorithms and the creation of comprehensive datasets tailored for SNNs. These efforts should focus on improving the usability and accessibility of SNN simulators to facilitate their adoption in various applications.

This study provides valuable guidance for selecting an appropriate open-source SNN simulator depending on its specific requirements and objectives. By advancing research in this field and addressing existing challenges, SNNs can emerge as powerful tools for intelligent edge computing and image processing applications.

This version merges the conciseness of mentioning various metrics from your abstract with the specific details from mine regarding CNN compatibility and the call to action for future research. It also emphasizes the real-world application focus on image recognition.

VII. Future work

In our pursuit to implement the algorithm for the CIFAR10-DVS and DVS-Gesture dataset, we encountered a significant hurdle during the training phase: a RAM crash resulting from the immense size of the dataset. This challenge severely impacted training efficiency, leading to prolonged training times and increased computational resource requirements.

Furthermore, our future work will prioritize the exploration of algorithmic enhancements aimed at improving memory utilization during training. We want to create novel model architectures or refine existing algorithms to streamline memory access patterns and minimize storage requirements. This may involve optimizing data structures, using batch processing techniques, or incorporating memory-efficient algorithms.

By overcoming these challenges and implementing algorithmic enhancements, we aim to enhance the efficiency and scalability of our SNN-based pattern recognition and image classification systems. These efforts align with our broader research goals of developing robust and efficient neuromorphic computing solutions for real-world applications.

Acknowledgement

This work was carried out under the guidance of Dr. Devarshi Mrinal Das, whose expertise and insights were invaluable throughout the entire process of developing this project. We are deeply grateful to him for the mentorship and support provided to us in completing this project report.

References

[1] O. Krestinskaya, A. P. James and L. O. Chua, "Neuromemristive Circuits for Edge Computing: A Review," in IEEE Transactions on Neural Networks and Learning Systems, vol. 31, no. 1, pp. 4-23, Jan. 2020, doi: 10.1109/TNNLS.2019.2899262.

[2] E. Chicca, F. Stefanini, C. Bartolozzi, and G. Indiveri, "Neuromorphic Electronic Circuits for Building Autonomous Cognitive Systems," in Proceedings of the IEEE, vol. 102, no. 9, pp. 1367-1388, Sept. 2014, doi: 10.1109/JPROC.2014.2313954.

[3] S. K. Bose, J. Acharya, and A. Basu, "Is my Neural Network Neuromorphic? Taxonomy, Recent Trends and Future Directions in Neuromorphic Engineering," 2019 53rd Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 2019, pp. 1522-1527, doi: 10.1109/IEEECONF44664.2019.9048891.

[4] J. K. Eshraghian et al., "Training Spiking Neural Networks Using Lessons From Deep Learning," in Proceedings of the IEEE, vol. 111, no. 9, pp. 1016-1054, Sept. 2023, doi: 10.1109/JPROC.2023.3308088.

[5] Samadzadeh, Ali, et al. "Convolutional spiking neural networks for spatio-temporal feature extraction." Neural Processing Letters 55.6 (2023): 6979-6995.

[6] Taylor, Luke, Andrew King, and Nicol Harper. "Robust and accelerated single-spike spiking neural network training with applicability to challenging temporal tasks." arXiv preprint arXiv:2205.15286 (2022).

[7] Perez-Nieves, Nicolas, and Dan Goodman. "Sparse spiking gradient descent." Advances in Neural Information Processing Systems 34 (2021): 11795-11808.

[8] Eshraghian, Jason K., et al. "Training spiking neural networks using lessons from deep learning." Proceedings of the IEEE (2023).

Appendix

GitHub repository link: https://bit.ly/3UtkceX