SpikeFI: A Fault Injection Framework for Spiking Neural Networks

Theofilos Spyrou, Said Hamdioui, Haralampos-G. Stratigopoulos

SpikeFI: A Fault Injection Framework for Spiking Neural Networks

Theofilos Spyrou*, Said Hamdioui* and Haralampos-G. Stratigopoulos†
*Computer Engineering Lab, Delft University of Technology, Delft, The Netherlands
†Sorbonne Université, CNRS, LIP6, Paris, France

***Abstract*—Neuromorphic computing and spiking neural networks (SNNs) are gaining traction across various artificial intelligence (AI) tasks thanks to their potential for efficient energy usage and faster computation speed. This comparative advantage comes from mimicking the structure, function, and efficiency of the biological brain, which arguably is the most brilliant and green computing machine. As SNNs are eventually deployed on a hardware processor, the reliability of the application in light of hardware-level faults becomes a concern, especially for safety- and mission-critical applications. In this work, we propose *SpikeFI*, a fault injection framework for SNNs that can be used for automating the reliability analysis and test generation. *SpikeFI* is built upon the SLAYER PyTorch framework with fault injection experiments accelerated on a single or multiple GPUs. It has a comprehensive integrated neuron and synapse fault model library, in accordance to the literature in the domain, which is extendable by the user if needed. It supports: single and multiple faults; permanent and transient faults; specified, random layer-wise, and random network-wise fault locations; and pre-, during, and post-training fault injection. It also offers several optimization speedups and built-in functions for results visualization. *SpikeFI* is open-source and available for download via GitHub at https://github.com/SpikeFI.**

***Index Terms*—Neuromorphic Computing, Spiking Neural Networks, Reliability, Fault Simulation, Testing, Fault Tolerance.**

I. Introduction

Neuromorphic computing is an emerging computing paradigm that has its roots in mimicking the spike-based operation of neurons in the biological brain. A neuromorphic processor essentially maps a Spiking Neural Network (SNN). SNNs can offer orders of magnitude more energy efficiency and inference speed compared to the more conventional Artificial Neural Networks (ANNs) [1], [2]. For this reason, SNNs open exciting new possibilities for realizing the next-generation Artificial Intelligence (AI) systems and for powering intelligent and autonomous edge devices with local AI processing. A major leap forward in the recent years is the development of several large-scale neuromorphic processors, e.g., SpiNNaker [3], TrueNorth [4], Loihi [5], BrainScaleS [6], and Neurogrid [7], supported also with software frameworks.

In this work, we address the dependability aspects of neuromorphic processors in view of the rare yet inevitable hardware-level faults. Hardware-level faults include bit-flips caused by cosmic ray particle strikes (a.k.a. soft errors) and defects and process parameter variations that are induced during manufacturing or occur in the field due to silicon aging mechanisms. SNNs show a large degree of inherent fault tolerance thanks to the analogy to the biological brain that has remarkable fault tolerance capabilities. Thus, most faults end up being benign: they are masked, i.e., their effect is not propagated to the output, or they can be tolerated, i.e., the output changes but the cognitive decision is still correct. However, there exist critical faults that will cause a wrong output disrupting the application.

More specifically, we propose a generic Fault Injection (FI) tool, named *SpikeFI*, for automating fault analysis of SNNs. Starting with an SNN model, the user is able to inject faults on different locations in the SNN architecture and assess their impact on the success of training and the accuracy of inference. The tool supports any SNN model, i.e., fully-connected. convolutional, or recurrent. It embeds a comprehensive fault model library that can be customized by the user. It supports single or multiple faults, transient or permanent faults, as well as statistical fault injection layer-wise and network-wise. Fault injection can be performed before, during or after training. *SpikeFI* also offers several simulation speedup options, such as early stop and late start, and has built-in various types of results visualisation functions.

SpikeFI has several use cases:

1) Understand the vulnerability of the SNN application to faults.
2) Assess how architectural choices (i.e., depth, layer size, feature map size, weight quantization, network compression, etc.) and the different per-layer hyper-parameters (i.e., neuron threshold, leakage, and refractory period) affect the resilience to faults so as to make early design decisions with reliability in mind.
3) Guide test generation algorithms aiming at generating test inputs for sensitizing and detecting critical faults [8]–[11]. Compact test sets can be used for post-manufacturing testing or can be replayed in idle times or periodically for in-field on-line testing.
4) Evaluate training algorithms in terms of their fault tolerance capabilities and develop fault-aware training algorithms [12]. For example, faults can be inserted during training, transiently across the epochs, to increase the robustness of the network to faults once deployed. Once the training is over the faults are ejected.
5) Assess the criticality of faults towards developing cost-effective hardware-level fault tolerance techniques [13]–[20].

SpikeFI performs fault analysis at the application level in software. Fault injection can be performed instead at the hardware description level, i.e., RTL, gate-level or transistor-level, but this requires the availability of the hardware im-

plementation and the more detailed the hardware description is, the lengthier the simulation time is. Already at RTL the simulation becomes intractable for sizeable SNN models given that the fault space explodes and that the impact of each fault is evaluated by performing inference on the complete testing set. Fault injection can also be performed on the actual hardware [21], but at this stage it may be too late to make any architectural changes for implementing fault tolerance techniques. To this end, *SpikeFI* adopts the flexibility and speed of software-level fault injection while supporting hardware-aware fault models by mapping them to software operators and accelerating faulty SNN instances on a GPU.

SpikeFI is publicly available and downloadable at https://github.com/SpikeFI. It is open-source and extendable, allowing researchers to implement their own fault models and results analysis.

There exist several works that have employed custom-made FI frameworks for SNNs (for example, see [8], [10], [12], [18], [19], [22]), but none of these FI frameworks was made publicly available and open-source. Very recently, the *SpikingJet* fault injector for SNNs was made publicly available [23]. *SpikingJet* is built on top of the SnnTorch framework [24], whereas *SpikeFI* is built on top of the Spike Layer Error Reassignment in Time (SLAYER) framework [25]. Compared to *SpikingJet*, *SpikeFI* provides several simulation speedup options, it supports in addition transient faults and fault injection before training, and it offers results visualization.

Software-level FI frameworks have also been developed for ANNs [26]–[30]. Recent efforts aim at improving the one-to-one mapping between hardware and software fault injection [31], [32], reproducing more complex fault models, i.e., extracted from radiation tests [33], or speeding up the analysis by reducing the fault injection space [34], [35] or the fault simulation time [36], [37]. Another possibility is to use generic FI tools [38], [39] to emulate fault effects in the hardware platform, i.e., GPU, running the application. Such FI frameworks are crucial towards the testability and dependability of AI hardware accelerators [40]–[44].

The rest of the article is structured as follows. In Section II, we provide background information on SNNs. In Section III, we discuss fault modelling for SNNs used by *SpikeFI*. The *SpikeFI* framework is presented in Section IV. The results are presented in Section V. Section VI concludes this article.

II. Background Information on SNNs

A. Principle of operation

Neural network models are classified into three generations. The first generation was based on McCulloch-Pitts neurons, also referred to as perceptrons, which give a digital output. The second generation of models applied an activation function to the output of the neurons, such as a rectified linear unit (ReLU) or sigmoid, and, in this way, they supported analog computation and learning algorithms based on gradient-descend, such as backpropagation. SNNs are the third generation [45], distinguishing themselves from their predecessors by their ability to mimic more realistically the biological brain. However, they constitute simplified models of their complex biological counterparts maintaining some of their aspects, since they are primarily used for computational purposes, rather than simulating the human brain.

Inspired from biological neural systems, SNNs encode the information in the timing of single action potentials, or spikes, and incorporate the time between successive spikes as a source of computation and communication among their spiking neurons. Spikes correspond to events generated whenever a change occurs providing a continuous time processing with very detailed time resolution reaching the micro- or nanosecond scale.

From a hardware perspective, SNNs form the basis of neuromorphic computing. Spiking neurons operate asynchronously to each other, as they are only utilized when an incoming spike stimulates them via their pre-synaptic connections. This makes SNNs event-based computation systems, which offers low latency and energy consumption compared to level-based ANNs. However, there are still challenges facing SNNs, such as the complexity of training.

Another characteristic of SNNs is the input type, which needs to be in a spiking form as well, i.e., the network is fed with a continuous-time event flow instead of static frames. A natural way to achieve this is with a neuromorphic camera, also known as Dynamic Vision Sensor (DVS). A DVS resembles the retina of the human eye and is composed of pixel-neurons that react to changes in brightness. When a sufficient change has occurred to the brightness of a pixel-neuron, it generates a positive or negative event, depending on the polarity of the change.

B. Spiking neuron models

There exist several spiking neuron models, ranging from biologically detailed ones, such as the Hodgkin-Huxley, to simplified ones that are more computational efficient for large networks and hardware implementation while still incorporating the main neuronal dynamics, such as the Integrate & Fire (I&F) [46]. As *SpikeFI* is built on top of SLAYER [25], and given that SLAYER employs the Spike Response Model (SRM) which is a generalization of the I&F model, herein we discuss in detail the SRM.

In the SRM, the state of the neuron at any given time is described by its membrane potential u. At its resting state, the membrane potential is set to a low value u_{rest}. The neuron integrates the incoming spikes from the synapses at its input and the membrane potential is increased or decreased according to the spike polarity. Once the potential reaches a certain threshold θ, the neuron fires a spike, which is propagated to the next layer of neurons via the synapses connected to its output, and the neuron is reset to its resting state again. At the same time, the neuron is regulated to not fire again for a while. The minimum time in-between successive spikes is called refractory period.

To mathematically express the above functionality, the SRM considers that the action of a neuron at any given time is a response to both the incoming activity and the neuron's own output. For this purpose, two response functions are used, namely, the synaptic kernel ϵ and the refractory kernel η. The

synaptic kernel ϵ describes the effect of an incoming spike train on the membrane potential and distributes the effect of the most recent incoming spikes on future output spike values, hence introducing temporal dependency. The refractory kernel η incorporates the effect of the neuron's own spike train onto its membrane potential. Two functions used to represent kernels ϵ and η are:

$$\epsilon(s) = \frac{s}{\tau_s} \cdot e^{1-\frac{s}{\tau_s}} \cdot H(s) \tag{1}$$

$$\eta(s') = -2\theta \cdot \frac{s'}{\tau_{ref}} \cdot e^{1-\frac{s'}{\tau_{ref}}} \cdot H(s'), \tag{2}$$

where $H(\cdot)$ is the unit step function, τ_s is the membrane time constant, τ_{ref} is the refractory time constant, s is the time passed since the last pre-synaptic spike at the input of the neuron, and s' is the time passed since the last post-synaptic spike at the output of the neuron.

The input and output spike trains of the neuron are denoted by $S_i(t)$ and $S_o(t)$, respectively:

$$\begin{aligned} S_i(t) &= \sum_f \delta(t - t_i^f) \\ S_o(t) &= \sum_f \delta(t - t_o^f), \end{aligned} \tag{3}$$

where δ is the Kronecker delta function to denote a spike and t_i^f and t_o^f represent the f-th firing time at the i-th neuron input and neuron output, respectively.

The membrane potential is expressed as:

$$u(t) = \sum_i \omega_i (\epsilon * S_i)(t) + (\eta * S_o)(t) + u_{rest}, \tag{4}$$

where ω_i is the weight of the synapse driving the i-th neuron input and $*$ is the convolution product.

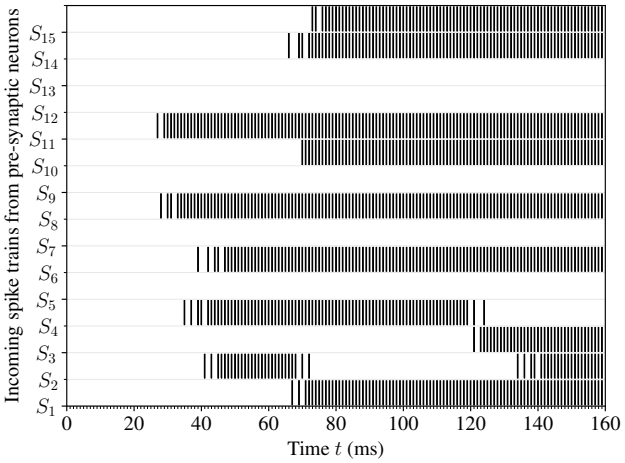
Assuming that the neuron has fired F spikes so far, a new spike $\delta(t - t_o^{F+1})$ is fired at time t_o^{F+1} when the neuron's membrane potential reaches the threshold θ and is appended to the output spike train as follows:

$$\begin{aligned} S_o(t) &= \sum_{f=1}^{F} \delta(t - t_o^f) + \delta(t - t_o^{F+1}) \\ t_o^{F+1} &= \min\{t : u(t) = \theta, t > t_o^F\}. \end{aligned} \tag{5}$$

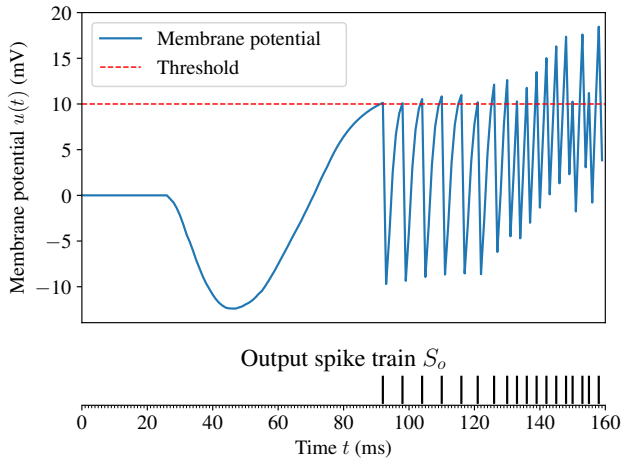
Once this happens, the neuron's membrane potential is reset to its resting state:

$$u(t_o^{F+1}) = u_{rest}. \tag{6}$$

Fig. 1 shows a simulation of a neuron following the SRM. The neuron is receiving input from many neurons. The pre-synaptic spike trains from 20 out of these neurons are shown in Fig. 1a. Fig. 1b shows the evolution of the neuron's membrane potential and its output spike train.



(a) Input spiking activity.



(b) Membrane potential and output spike train.

Fig. 1: SRM simulation.

C. Training schemes

Training large SNN models remains a challenge [1], [2], [47]. The classic backpropagation algorithm used in ANNs cannot be applied directly to work with spiking events due to their non-differentiable nature. Several learning schemes have been developed to overcome this challenge, including the biologically inspired Spike-Timing-Dependent Plasticity (STDP), training an ANN and converting it to a SNN, evolutional algorithms, and spike-based backpropagation. Spike-based backpropagation is currently one of the most practical and accurate techniques for training SNNs. SLAYER [25] trains a SNN with a variation of backpropagation using the probability of a spiking neuron to change state, i.e., fire a spike or move back to its resting state. The fact that SNNs can have the same topologies as ANNs, e.g., fully-connected, convolutional, recurrent, etc., allows for embedding SLAYER in already mature Machine Learning (ML) frameworks with some adjustments to add support for the spiking functionality. To this end, there is a PyTorch version of SLAYER that can be used for both the training and the inference of any SNN

TABLE I: Behavioral-level representations of built-in fault models.

Fault type		Fault model	Fault effect
Neuron faults	Hard	Dead neuron	$\hat{S}_o(t) = 0$
		Saturated neuron	$\hat{S}_o(t) = \sum_{n=0}^{\infty} \delta(t-n)$
		Stuck-at-x neuron	$\hat{S}_o(t) = x \cdot \sum_{n=0}^{\infty} \delta(t-n)$
	Parametric	Integration fault	$\hat{\tau}_s = \rho \cdot \tau_s$
		Refractory fault	$\hat{\tau}_{ref} = \rho \cdot \tau_{ref}$
		Threshold fault	$\hat{\theta} = \rho \cdot \theta$
Synapse faults	Hard	Dead synapse	$\hat{\omega}_i = 0$
		Saturated synapse	$\hat{\omega}_i >> \omega_i$ or $\hat{\omega}_i << \omega_i$
	Parametric	Perturbed synapse	$\hat{\omega}_i = \rho \cdot \omega_i$
		Bit-flipped synapse	$\hat{\omega}_i = Q^{-1}(Q(\omega_i) \oplus 2^b)$

model. *SpikeFI* is built upon SLAYER and PyTorch, inheriting their features and extending their capabilities to support FI experiments.

III. Fault Modeling

SpikeFI is a fault injector at the application level. Hardware-level faults are translated to behavioral-level faults which, thereafter, are reproduced mathematically into the SRM described in Section II-B. We consider that the processing elements of the SNN, i.e., neurons and synapses, are discrete entities that can fail independently. A bottom-up approach can be followed to extract the spiking neuron and synapse faulty behaviors starting from transistor-level simulations [48]. *SpikeFI* conforms to the established practice and adopts all widespread and conventional fault models in the literature that are derived from various digital, analog, and mixed analog-digital SNN hardware implementations. These fault models are built-in within *SpikeFI* and are delivered as a library. The library is fully modifiable and extendable, i.e., developers are free to create their own fault models depending on the hardware implementation and fault occurrence probabilities.

Modeling faults at behavioral-level allows evaluating their impact without the need of knowing the details of their source or mechanism, avoiding in this way costly low-level simulations, i.e., at the transistor, gate, microarchitectural level, etc., performed at the network scale. Additionally, behavioral-level fault modeling provides the flexibility to model any possible hardware-level fault, as long as a mathematical formula describing it can be derived. Another advantage is that the results are not tied to a specific hardware accelerator design, thus the drawn conclusions tend to be more generic.

Next, we describe the built-in fault models, which are summarized in Table I, by separating them into neuron and synapse fault models. Fault models can be further divided into hard and parametric fault models, depending on whether the processing element presents an outright failure or deviation.

A. Neuron faults

1) Hard faults:

a) Dead neuron: A fault that halts the neuron's spiking activity and makes it unresponsive to any input. To model a dead neuron, its output spike train is set to zero, i.e., $\hat{S}_o(t) = 0$.

b) Saturated neuron: A fault that causes a neuron to be firing non-stop, even in the absence of input activity. A saturated neuron can be considered as the complementary extreme case of a dead neuron, where the output spike train is never zero, i.e., $\hat{S}_o(t) = \sum_{n=0}^{\infty} \delta(t-n)$.

c) Stuck-at-x neuron: A fault that causes the neuron's output to be stuck-at a value x, i.e., $\hat{S}_o(t) = x \cdot \sum_{n=0}^{\infty} \delta(t-n)$, where $x \in \mathbb{R}$. A stuck-at neuron can be viewed as the generic hard neuron fault, with the extreme dead and saturated neuron faults being derived by setting $x = 0$ and $x = 1$, respectively.

2) Parametric faults:

a) Integration fault: A fault that causes the membrane time constant τ_s of the synaptic kernel ϵ in Eq. (1) to be perturbed to a new value $\hat{\tau}_s = \rho \cdot \tau_s$, $\rho \in \mathbb{R}$, affecting the neuron's easiness to fire spikes. Depending on whether $\hat{\tau}_s > \tau_s$ or $\hat{\tau}_s < \tau_s$, the kernel function ϵ allows for an easier or harder integration, respectively, or, conceptually speaking, makes the neuron more or less sensitive towards incoming spikes at its input.

b) Refractory fault: A fault that causes the refractory time constant τ_{ref} of the refractory kernel η in Eq. (2) to be perturbed to a new value $\hat{\tau}_{ref} = \rho \cdot \tau_{ref}$, $\rho \in \mathbb{R}$, affecting the neuron's refractoriness, i.e., the minimum time that needs to elapse before the neuron is capable of firing again. If $\hat{\tau}_{ref} > \tau_{ref}$, then the refractory kernel η converges slower to zero, meaning that the refractoriness of the neuron becomes stronger. On the other hand, if $\hat{\tau}_{ref} < \tau_{ref}$, then the neuron's state is less tightly associated to its own output activity and, thereby, it demonstrates a weak refractoriness.

c) Threshold fault: A fault that causes the threshold of the neuron θ to be perturbed to a new value $\hat{\theta} = \rho \cdot \theta$, $\rho \in \mathbb{R}$. As it can be inferred from Eq. (5), a faulty threshold $\hat{\theta}$ affects the output spike train either in a contributory or in a suppressive way, depending on whether $\hat{\theta} < \theta$ or $\hat{\theta} > \theta$, respectively. Similarly to an integration fault, a lower threshold leads to a neuron that fires easier if excited by the same stimuli, as the membrane potential reaches the threshold faster. This fault has a second effect since the threshold is also used in the refractory kernel η. Therefore, from Eq. (2), an increase in the threshold's value, implies a higher refractory period, thus taking the neuron longer to recover after firing, while a decrease has the reverse effect.

B. Synapse faults

1) Hard Faults:

a) Dead synapse: A fault that holds the synaptic weight to zero, i.e., $\hat{\omega}_i = 0$, disabling the synapse and "cutting" the connection between the pre- and post-synaptic neurons.

b) Saturated synapse: A fault that saturates the synaptic weight to an extreme positive $\hat{\omega}_i >> \omega_i$ or extreme negative $\hat{\omega}_i << \omega_i$ value. Some representative positive and negative saturation values could be respectively the highest and lowest values of the weight distribution resulting from training.

2) Parametric Faults:

a) Perturbed synapse: A fault that perturbs the synaptic weight to a new value $\hat{\omega}_i = \rho \cdot \omega_i$, $\rho \in \mathbb{R}$.

b) Bit-flipped synapse: From a hardware perspective, representing synaptic weights as digital words and storing them in on-die memories is common practice. This fault flips one or more bits of a N-bit representation of the synaptic weight. For example, for a N-bit integer representation, to model this type of fault, the real value of the synaptic weight is quantized with a precision of N bits, the selected bits are flipped, and then the weight is converted back to a real number. At the end, the faulty value of the synaptic weight is given by $\hat{\omega}_i = Q^{-1}(Q(\omega_i) \oplus 2^b)$, where Q is the quantization function, Q^{-1} is the inverse quantization function, $\oplus$ is the logic XOR operator, and b represents the position(s) of the flipped bit(s), where $b = 0$ corresponds to the Least Significant Bit (LSB). This fault model can assume a Bit Error Rate (BER) probability for the memory storing the synaptic weights of the network in a binary data type.

C. Permanent and transient fault effect

Depending on the nature of a fault and the factors that led to its occurrence, its effect may be either permanent or transient. SNNs have a global internal clock that defines the discrete times when neurons can be firing. If a fault is transient, its effect is active for a limited period or number of clock cycles. For the rest of the FI experiment time, the components affected by the transient fault are restored to their nominal state and, therefore, the original behavior is expected. For example, denoting the transient fault duration by $[t_1, t_2]$, the behavioral description of a transient neuron saturation fault is:

$$\hat{S}_o(t) = \begin{cases} \delta(0) & : \quad t \in [t_1, t_2] \\ S_o(t) & : \quad \text{otherwise} \end{cases} \tag{7}$$

and the behavioral description of a perturbed synapse fault is given by:

$$\hat{\omega}_i(t) = \begin{cases} \rho \cdot \omega_i & : \quad t \in [t_1, t_2] \\ \omega_i & : \quad \text{otherwise} \end{cases} . \tag{8}$$

IV. The *SpikeFI* Framework

A. Overview and supported features

SpikeFI is a native PyTorch framework [49] built upon the SLAYER framework [25], extending the capabilities of SLAYER to add support for setting and automatically executing FI and reliability analysis experiments. SLAYER is a notable training framework for SNNs that is contributing to the growing interest in SNNs. It is developed to enable efficient spike-based backpropagation learning for deep SNNs. It has been incorporated into Lava, Intel's open-source software framework for developing SNNs. *SpikeFI* employs the same principles and programming concepts and paradigms as its underlying frameworks PyTorch and SLAYER preserving compatibility. Any arbitrary SNN model implemented in SLAYER can be the subject of FI and reliability analysis, without needing to make any modifications to the model. Researchers and developers already familiar with SLAYER can therefore jump directly into using *SpikeFI*. *SpikeFI* is offered as open-source software and is available for download and to contribute via the GitHub platform: github.com/SpikeFI/.

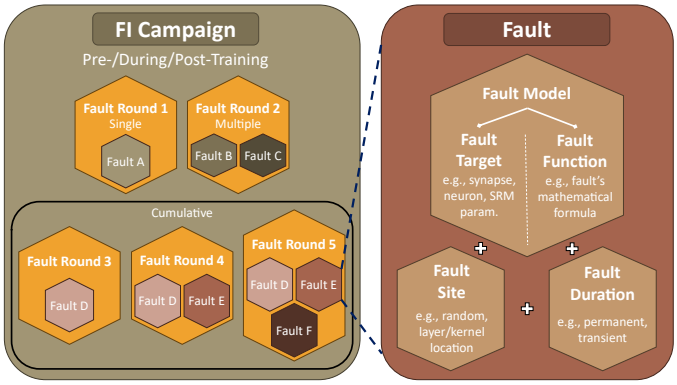


Fig. 2: The organization of an FI campaign.

SpikeFI supports the following features and scenarios:

1) *Flexible fault modeling scheme*: *SpikeFI* has an integrated comprehensive library of predefined fault models to select from, as described in Section III. Faults can be injected into any processing element, i.e., neuron or synapse, and at different levels, i.e., isolated processing element, layer-wise, and network-wise. *SpikeFI* being open-source allows the user to design and integrate custom fault models.
2) *Pre-/during/post-training FI injection*: *SpikeFI* allows injecting faults before, during, or after the training phase prior to inference. The motivation is different across these scenarios. In pre-training fault injection, the goals can be to perform fault-aware training, re-train the network after the occurrence of a critical fault, and, in general, to study the network's capability to learn around faults. During training fault injection aims at studying the effect of faults occurring while training is progressing. This is useful when training deep SNNs as it can take significant time during which the hardware can suffer a fault. Post-training fault injection aims at studying the effect of faults on the inference accuracy. The analysis here aims at studying the inherent reliability and deriving critical fault types and locations. This information can subsequently be used for developing cost-effective test and fault tolerance strategies.
3) *Multi-round FI campaign*: A FI campaign may be composed of multiple experiments that are conducted sequentially independent of each other.
4) *Single/multiple/cumulative FI injection*: Each of the fault rounds may contain a single fault, multiple faults or accumulated faults. In the latter scenario, in each FI experiment the set of faults is increased to observe the accumulative effect of the new faults added.
5) *Permanent/transient fault analysis*: A fault can be designed to be permanent or transient of varying duration, as described in Section III-C.
6) *Optimization options*: *SpikeFI*, being built on top of PyTorch, utilizes GPU acceleration. It also offers various optimization options to speedup fault injection experiments, such as proper *for*-loop ordering, late start, early stop, and batch-wise inference, which will be described in detail in

Section IV-D.

7) *Results visualization*: *SpikeFI* optionally saves spike trains for off-line analysis, but it also offers several built-in results visualization functions based on the inference accuracy drop metric, as it will be discussed in Section IV-E3 and will be demonstrated in Section V-C.

B. Structuring of a FI experiment

Fig. 2 summarizes the hierarchical organization of programming elements that shape a FI campaign in *SpikeFI*.

1) Fault Round: A group of faults to be injected altogether into the network. A fault round can contain a single or multiple faults. The collective effect of all faults belonging to the same round is evaluated simultaneously in a single inference. *SpikeFI* also offers the possibility to perform cumulative fault analysis, i.e., define multiple fault rounds where each fault round contains the faults of the previous fault round plus some additional faults. Once a fault round has been evaluated, the faults are withdrawn from the network before continuing to the next fault round.

2) Fault: The actual fault to be injected into a network, composed of a fault model, one or more fault sites, and a fault duration.

3) Fault model: The fault model is in turn composed of a fault target and a fault function.

a) Fault target: The fault target can be: (i) the neuron output; (ii) the SRM parameters, i.e., neuron's membrane time constant, threshold, and refractory time constant; and (iii) the weight of a synapse, as described in Sections III-A and III-B.

b) Fault function: The fault function is the mathematical formula that describes how the fault is to affect the operation of the targeted processing element, as described in Sections III-A and III-B.

4) Fault Site: The fault site is the location of the fault within the network. The fault can be isolated affecting a specific processing element or it can be applied to multiple processing elements simultaneously. The user has also the option to create random fault sites per layer or across the network following some statistical fault distribution. The site of a fault is composed of the layer and coordinates of the processing element within the layer. In the case of a neuron, the site is the quadruplet (l, c, x, y), where l is the layer name or number, c is the feature map number within the layer, and (x, y) are the coordinates of the neuron within the feature map. In the case of a synapse, the site is defined by the quadruplets of the two neurons connected by the synapse.

5) Fault Duration: The fault duration provides information on when the fault is activated and for how long it remains active, as described in Section III-C. By default, a permanent fault means that is active for the whole duration of the input sample, whereas the duration of a transient fault is only a portion of the duration of the input sample.

C. FI implementation into SLAYER

To inject a fault, *SpikeFI* modifies the PyTorch computation flow in SLAYER. For every fault model, there is a suitable modification to be applied, as illustrated in Fig. 3. To apply

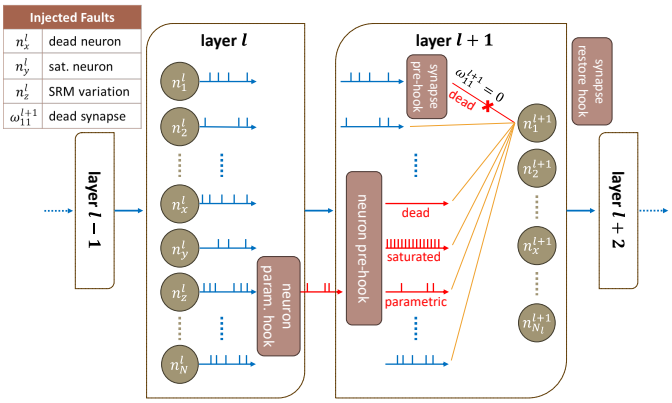


Fig. 3: FI implementation into SLAYER.

the modification, *SpikeFI* makes use of PyTorch pre-hook and hook functions, i.e., a function called right before and right after the evaluation of a module, respectively.

Let us first consider neuron hard faults in layer l. The fault function of the fault model returns the output of the faulty neurons and updates the output of layer l for these neurons while the rest of the neurons retain their fault-free output. Then, the output of layer l is propagated to the input of the subsequent layer $l+1$ of the network. This is implemented with a neuron pre-hook function attached to the input of layer $l+1$, as illustrated in Fig. 3.

Regarding neuron parametric faults, in SLAYER the SRM parameters are set globally for a layer and are shared among its neurons, thus it is not possible to modify the parameters only for a subset of neurons. For this reason, *SpikeFI* uses a hook and a pre-hook function. The first one, called neuron parametric hook, receives the same input as the faulty layer l, creates a "dummy" copy of the layer with altered SRM parameter for all neurons according to the fault model, repeats the parts of the calculation on this "dummy" copy that concern the affected SRM parameters, and feeds the result to the next neuron pre-hook function attached to the input of layer $l+1$. The neuron pre-hook function selects only the faulty neuron(s) and replaces their output spike train with the spike train of the corresponding neuron(s) in the dummy layer.

Regarding the synaptic weights, *SpikeFI* uses a synapse pre-hook function to alter the synaptic weight value according to the fault model prior to the forward pass through the faulty layer. At the end of the faulty layer evaluation, there is a synapse restore hook function that restores the original synaptic weight, so that there is no interference between successive fault rounds.

D. Optimizations

1) Ordering of nested for loops: All fault rounds are evaluated by performing inference for the same set of input samples, which could correspond to the complete testing dataset or part of it. Essentially, a FI campaign is a nested loop iterating over all fault rounds and over the set of input samples. The ordering of the two *for* loops has in fact an effect on the FI campaign runtime. *SpikeFI* places the input samples in the outer *for* loop and the fault rounds in the inner *for* loop since this results

in faster runtime, as it will be demonstrated quantitatively in Section V-B. The underlying reason is that the alternative *for*-loops ordering would require transferring the dataset to the GPU for the computation multiple times, equal to the number of fault rounds, which would add a significant time overhead. Instead, with the selected *for*-loop ordering, a batch transferred to the GPU is reused for all fault rounds, thus circumventing this time overhead.

2) Late Start: As typically there are multiple fault rounds in a FI campaign, there is significant repetition of forward passes through initial fault-free layers of the network. For example, consider two fault rounds with a single fault each, located at layers i and $j > i$, respectively. The forward pass is repeated twice up to layer $i-1$ which is redundant. To save simulation time, for every fault round, ***SpikeFI*** uses the late start option that skips layer computations up to the leftmost faulty layer, denoted by l_{left}.

For this purpose, in a preparatory phase prior to the FI experiment, ***SpikeFI*** performs a nominal inference for the complete testing dataset and records the deterministic golden output of all layers. More formally, for each layer l, ***SpikeFI*** computes the matrix A^l of its output spike trains with dimensions $N^l \times d$, where N^l is the number of flattened neurons in layer l and d is the number of timestamps within the inference window. The SNN has a global clock with period T. The timestamps are denoted by $t_j = j * T$, $j = 1, \cdots d$. $A^l(i,j) = 1$ if neuron i in layer l fires at timestamp t_j, otherwise $A^l(i,j) = 0$. The layer matrices A^l are combined to generate the network matrix $A = [A^1, \cdots, A^L]$, where L is the number of layers.

During the FI experiment, for every fault round, ***SpikeFI*** first orders the faults based on the layer wherein they occur in ascending order. If the first faults appear at layer l_{left}, then ***SpikeFI*** uses the golden output of layer $l_{left}-1$ and continues the simulation from this point onward.

Note that if the fault round contains only hard neuron faults, then the simulation can continue from layer $l_{left}+1$. This is because of the implementation in PyTorch which injects the fault with a neuron pre-hook function attached to the input of the layer following the faulty layer. This means that late start can offer speeds-up even when the leftmost faulty layer is the first layer.

3) Early Stop: As mentioned in the introduction, many faults end up being benign. If in a fault round the output of the rightmost faulty layer, denoted by l_{right}, is unaffected matching the golden response, then it is pointless to continue the simulation as it will be like simulating a nominal fault-free network. ***SpikeFI*** uses the early stop option to skip this redundant computation. More specifically, early stop uses the golden layer output matrices A^l as in late start. Assuming a fault round with the last faults occurring in layer l, the early stop option computes the same matrix denoted by A^l_f, where the subscript f indicates a fault round, subtracts it from the golden matrix A^l to produce the matrix $B^l = A^l - A^l_f$, and computes the summation of all elements of B^l denoted with the elementwise 1-norm $\|B^l\|_1$. If $\|B^l\|_1 = 0$ it means that all faults in this fault round are benign and the simulation stops at layer l. If $\|B^l\|_1 > 0$ it means that with respect to the golden output there is a spike count or spikes timing difference.

SpikeFI also offers the possibility of considering a tolerance ϵ for the early stopping criterion. In this case, the simulation stops if $\|B^l\|_1 \leq \epsilon$. This tolerance should be used with care as it is likely that a fault induces a small spike count or spikes timing difference within the tolerance, yet this small difference is sufficient to cause the network's output spike trains to change to the point where the top-1 prediction changes. In this case, we stop early the simulation of a fault round that contains critical faults, mislabelling this fault round as benign.

E. Complete FI experiment flow

A FI experiment is divided into three stages, namely the preparation, execution, and results extraction and visualization stages.

1) Preparation stage: First, the validity of all faults in all fault rounds is checked. If a fault is invalid, for example the fault site is nonexistent, then the fault is dropped from the experiment. Then, random faults are assigned a random site. After these two steps, the faults within each fault round are ordered according to the layer they belong to in ascending order so as to identify the leftmost l_{left} and rightmost l_{right} layers in the fault round. The verified fault rounds together with their sorted list of faults is communicated back to the user to acknowledge the FI experiment setup. Lastly, for each batch and before the FI starts, ***SpikeFI*** performs an inference on the nominal network so as to record the golden outputs of all layers and form the matrices A^l used in the late start and early stop optimizations.

2) Execution stage: Fig. 4 shows the ***SpikeFI*** complete FI experiment flow including all optimizations. The dataset is transferred to the GPU in batches and for each batch the same FI experiment is performed in parallel for every input sample in the batch. The FI experiment is composed of a number of fault rounds simulated sequentially. For a given fault round, the simulation starts from the leftmost faulty layer l_{left} using the golden output of the previous layer, or from layer $l_{left}+1$ if l_{left} comprises only hard neuron faults. Inference continues up to the rightmost faulty layer l_{right}. In the case of a fault round with a single fault, l_{left} and l_{right} coincide. At this point, if the early stop criterion is met then the simulation stops and we proceed to the next fault round. Otherwise, the simulation continues up to the last layer. Before the next fault round starts, the results are saved and the network is initialized back to its original fault-free state.

3) Results extraction and visualization stage: Based on the spike encoding method employed by the SNN, i.e., rate coding, temporal coding, etc., for each fault round, ***SpikeFI*** uses the output spike information in the matrices A^L and A^L_f to report the accuracy of the faulty network. For example, let us assume rate encoding, which is the most widely used spike encoding method, and a classification cognitive task. In this case, the output layer comprises one neuron per class and the winning class, i.e., the top-1 prediction, is that whose neuron fires the largest number of spikes within the inference time window. These spike counts are computed using the matrices A^L and

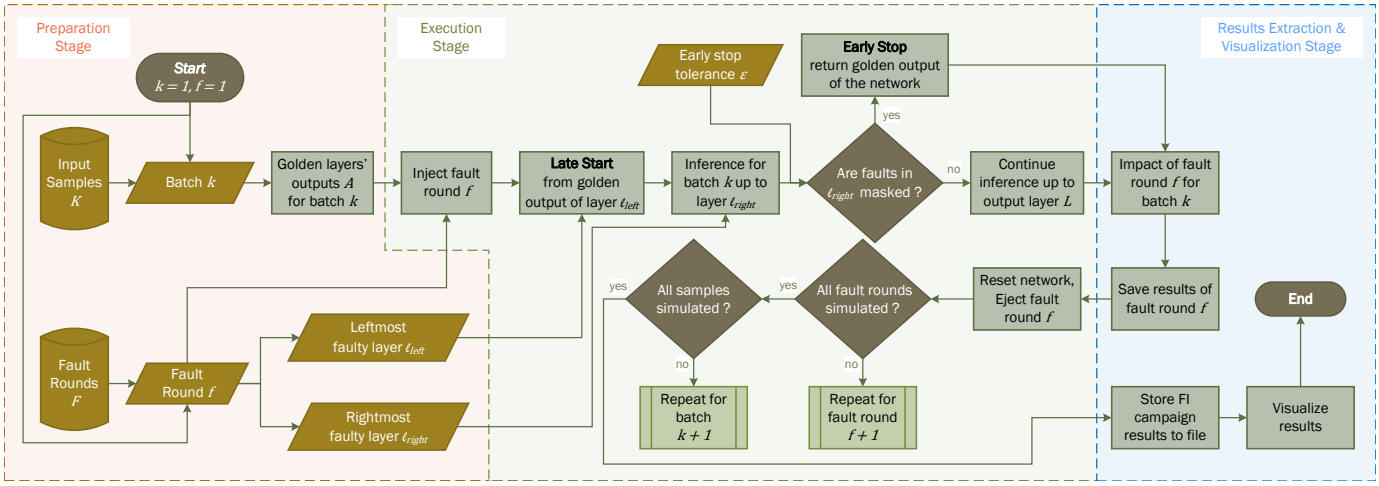


Fig. 4: Flowchart of a FI campaign in *SpikeFI*.

A_f^L to assess if the faulty network changes the top-1 prediction of an input sample which contributes to accuracy loss. The user can define a misprediction tolerance value. If the accuracy drop is larger than this tolerance value, then this signifies a critical fault round. In this way, fault rounds are labeled as critical or benign. Optionally, the matrices A^l and A_f^l, or their last parts A^L and A_f^L, can be saved for offline analysis by the user. Finally, *SpikeFI* reports the runtime to complete the FI campaign.

SpikeFI comes with several built-in results visualization functions that work for any fault model, as it will be shown in Section V. For example, misprediction rates can be plotted for isolated and random faults in the form of bar plots and heat maps and for parametric faults as a function of the parameter deviation. Results can also be presented bit-wise and layer-wise so as to perform comparisons.

Algorithm 1: Example of a FI campaign in *SpikeFI*.

Data: $net, shape\_in, slayer, test\_set$
$cmpn \leftarrow Campaign(net, shape\_in, slayer)$
$f_x \leftarrow Fault(DeadNeuron(), FaultSite(SF2))$
$f_y \leftarrow Fault(SatuSynapse(10), FaultSite(SF1))$
$f_z \leftarrow Fault(ParamNeuron(theta, 0.5), 4)$
$cmpn.inject(f_x)$
$cmpn.then\_inject(f_y, f_z)$
$cmpn.run(test\_set)$
$cmpn.export()$
$cmpn.save()$
$bar(cmpn)$
$cmpn.eject()$
$cmpn.inject\_complete(BitflippedSynapse(7), SF2)$
$cmpn.run(test\_set)$
$heat(cmpn)$

F. Open source code and example

A detailed documentation is provided in the GitHub platform for the easy integration and usage of the *SpikeFI* framework. The structure of the *SpikeFI* framework is organized in the form of a Python package to be imported and use its functions immediately. There are four main modules within the Python package, namely, the *core*, *fault*, *models*, and *visual* modules, each containing relative classes and functions to implement the framework's functionalities. An additional *demo* module is included with the implementations of the two SNN architectures of Section V-A and example scripts to showcase the usage of *SpikeFI* in various scenarios.

Algorithm 1 presents a pseudo-code example resembling Python syntax of two FI campaigns. First, the campaign object $cmpn$ is created and initialized with the network model net, a vector with the dimensions of the input data samples $shape\_in$, and the spiking-related information object $slayer$, which is provided by SLAYER and is initialized by the user. More specifically, $slayer$ contains information about the SRM parameters of the spiking neurons, the duration of the input data samples, the global clock period, the target number of spikes for the winning class neuron, etc. Next, the faults f_x, f_y, f_z are defined corresponding respectively to a dead neuron, a positively saturated synapse with value 10, and a parametric neuron fault that decreases parameter θ to 50% of its nominal value. Single faults f_x and f_y are assigned a random fault site in layers $SF2$ and $SF1$, respectively. Multiple fault f_z is initialized with 4 random fault sites anywhere in the network. Fault f_x is injected to the network as the first single-fault fault round using the $cmpn.inject$ function. Then, faults f_y and f_z are included in a second five-fault fault round using the $cmpn.then\_inject$ function. The function $cmpn.inject$ always adds the faults to the current fault round, while $cmpn.then\_inject$ adds a new fault round. The next line calls the $cmpn.run$ function that takes as argument a reference to the complete testing set $test\_set$ and performs the preparation and execution stages of the FI campaign, as described in Sections IV-E1 and IV-E2. By default, $cmpn.run$ makes use of all available optimization options described in Section IV-D, but the user can opt to use late start or early stop. Once the FI campaign is over, the details of the FI experiment, i.e., fault rounds, and the results, i.e., matrix A_f^L, classification accuracy for each fault round, etc., are extracted

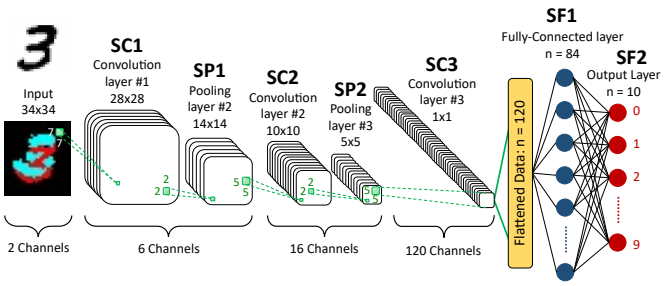


Fig. 5: N-MNIST SNN.

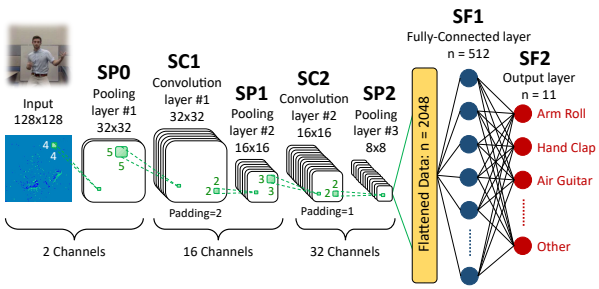


Fig. 6: IBM DVS128 Gesture SNN.

as a FI campaign data object using the function $cmpn.export$ and stored using the $cmpn.save$ function. Depending on the preferred results visualization, the user can choose among a set of plotting functions, as it will be demonstrated in Section V-C. The FI campaign data object is fed to the plotting functions to visualize the results. In this example, the results are plotted using the bar function. This FI campaign terminates by calling the $cmpn.eject$ function, which removes all fault rounds and re-initializes the network to perform a new FI campaign if desired, allowing for unlimited reuse of the campaign object $cmpn$. A second FI campaign is then executed using the function $cmpn.inject\_complete$, which completes the inject functions family. $cmpn.inject\_complete$ creates as many fault rounds with single faults as the number of processing elements in the specified layer. In this example, we perform a bit-flip in the MSB of an 8-bit integer representation for all synapses in layer $SF2$. This second FI campaign is executed with the $cmpn.run$ function and the results are visualized in a form of a heat map by calling the $heat$ function.

V. Results

A. Case Studies

SpikeFI is demonstrated on two convolutional SNNs trained to classify the N-MNIST [50] and IBM's DVS128 Gesture [51] datasets. These SNNs are modelled and trained in SLAYER, and are included in the *demo* package of the *SpikeFI* framework in GitHub. They use rate coding, i.e., the winning class is selected after the neuron at the output layer which is triggered the most producing the highest number of spikes.

The N-MNIST dataset is a neuromorphic, i.e., spiking, version of the MNIST dataset, which comprises images of handwritten arithmetic digits in gray-scale format [50]. It consists of 70000 sample images that are generated from the saccadic motion of a DVS in front of the original images in the MNIST dataset. The samples in the N-MNIST dataset have a duration of 300 ms. The dataset is split into a training set of 60000 samples and a testing set of 10000 samples. The SNN architecture, shown in Fig. 5, is a spiking version of the LeNet-5 architecture [52]. It consists of 3 convolutional layers with 2 2x2 sum-pooling layers in between them and 2 fully-connected layers at the end for the final decision of the network. The classification accuracy on the testing set is 97.8%.

The IBM's DVS128 Gesture dataset consists of 29 individuals performing 11 hand and arm gestures in front of a DVS,

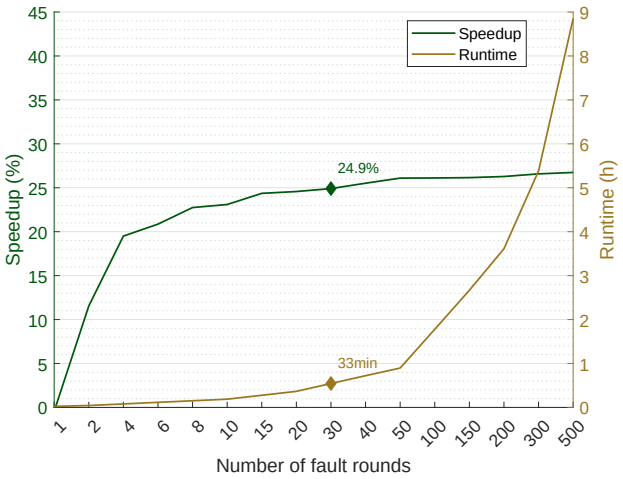


Fig. 7: Speedup and runtime when using fault rounds in the inner *for* loop.

such as hand waving and air guitar, under 3 different lighting conditions [51]. In total, the dataset comprises 1342 samples of duration 6 s, which is trimmed to 1.5 s to speedup simulation. The designed SNN, shown in Fig. 6, is an adaptation of the network proposed in [51]. It starts with a 4x4 sum-pooling layer to reduce the big size of the input samples. Next, there are 2 convolutional layers followed by a 2x2 sum-pooling layer each. The architecture is concluded with 2 fully-connected layers. The network performs with an 86.4% accuracy on the testing set, which is acceptable considering the shortened samples of the dataset and the shallower architecture compared to the architecture in [51].

B. Optimization speedups

Herein, we use the N-MNIST SNN as a benchmark for quantifying the speedup improvements when using the different optimizations.

For a fair comparison, all the FI experiments below were executed one at a time on the same system configuration composed of an Intel Xeon® W-2133 CPU and a NVIDIA Quadro® RTX 4000 GPU, with the system being reserved for the experiment.

1) Ordering of nested for loops: First, we show that placing the fault rounds in the inner *for* loop as opposed to the outer *for* loop speeds up the analysis. For this purpose, we perform successive FI campaigns with a batch size of 1 and

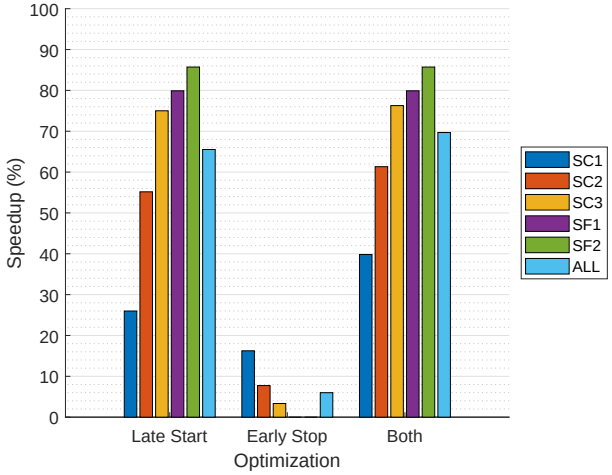


Fig. 8: Speedup using the late start and early stop optimizations.

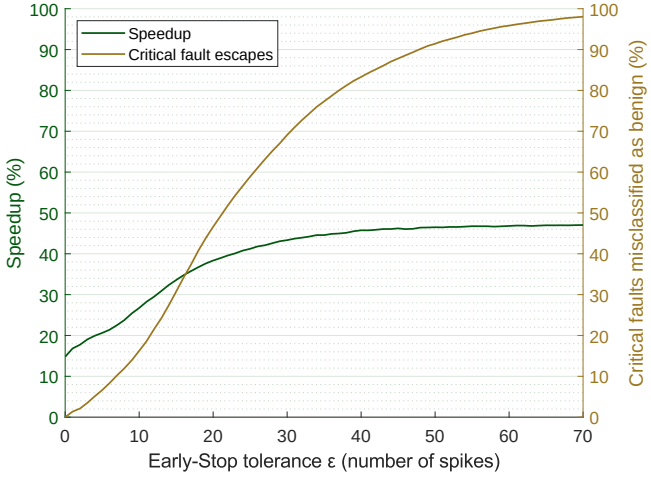


Fig. 9: Speedup and critical fault escape rate when using early stop with tolerance $\epsilon > 0$.

with increasing number of fault rounds, and we measure the speedup as well as the total runtime. Each fault round is composed of a single random dead neuron fault, and the faults in the successive fault rounds are accumulated, i.e., one fault round contains all faults of the previous fault round plus a new random one. For this experiment, the early stop and late start optimizations are disabled. The result is shown in Fig. 7. As it can be seen, the speedup increases exponentially with the number of fault rounds, with the speedup slowing down after 10 fault rounds and converging to around 27%. The runtime increases linearly with the number of fault rounds (note that the scale of the x axis is not linear). The convergence occurs when the runtime starts dominating, overshadowing the benefit from the reduced data transfers to the GPU. Based on this result and as discussed in Section IV-D1, *SpikeFI* places the fault rounds in the inner *for* loop.

2) Late start and early stop: To demonstrate the speedups offered by late start and early stop, we use a FI experiment with a batch size of 1 and 30 fault rounds, with each fault round containing a single random dead neuron fault. As marked in Fig. 7, such an experiment takes up approximately 33 min. The FI experiment is repeated layer-wise for all 5 layers by concentrating all 30 faults in one layer. For the last layer that has only 10 neurons, we triplicate each fault round. The FI experiment is also performed network-wise by distributing the fault rounds equally across the 5 layers, i.e., placing 6 faults per layer.

Fig. 8 shows the speedup for these layer-wise and network-wise FI experiments when using late start and early stop optimizations as stand-alone and combined. For the early stop we use $\epsilon = 0$. As expected, we observe that the speedup offered by late start improves as the leftmost faulty layer moves to the right. In contrast, early stop is more effective as the rightmost faulty layer moves to the left. As the fault model is dead neuron faults, late start can offer speedup even for faults in the first layer (see Section IV-D2).

Another observation is that late start offers significantly greater speedups compared to early stop. For the late start the speedup ranges from 26% to 86% moving from layer 1 to layer 5, while for early stop the speedup is less spectacular. It ranges from 16% to 3% moving from layer 1 to layer 3, while it vanishes for layers 4 and 5. For the network-wise FI experiment, late start offers a speedup of 66%, whereas the speedup for early stop is a modest 6%. Combining both the speedup reaches 70%. The reason behind late start being more effective than early stop is that early stop is activated only when a fault round is benign and also requires the evaluation of $\|B^l\|_1$ whose time can counterbalance the average speedup benefit. In contrast, late start is applied to any type of fault round and is activated instantaneously. Early stop can offer a significant speedup for the initial layers when a large percentage of faults are benign.

Fig. 9 depicts the effect of using $\epsilon > 0$ in early stop. For this experiment, the FI campaign is performed on the first layer, containing as many fault rounds as the number of neurons in this layer, each with a single dead neuron fault. We observe that as ϵ increases, the percentage of critical faults misclassified as benign increases. For this FI experiment, improving further the speedup is not possible without misclassifying critical faults. For $\epsilon = 1$, the speedup is around 15% while the misclassification moves away from zero. Therefore, early stop with tolerance $\epsilon > 0$ should be used with caution, as it could lead to masking the effect of critical faults.

3) Batched inference: Finally, Fig. 10 shows the speedup and average runtime per fault round as a function of batch size. The FI experiment is composed of 30 fault rounds of a single random dead neuron fault, distributed equally across the 5 layers. The late start and early stop options are turned on, using $\epsilon = 0$ for the early stop. The baseline speedup for batch size 1 is 70%, as shown in the last column of Fig. 8, where the exact same FI experiment is performed. As it can be seen from Fig. 10, the speedup increases by a maximum of approximately 15% for batch size 11, converging to around 85% for larger batch sizes, while the average runtime per fault round reaches a minimum of 6.1 s at this point. The reason behind this initial speedup improvement is that, on one hand, PyTorch is optimized to calculate more efficiently the output

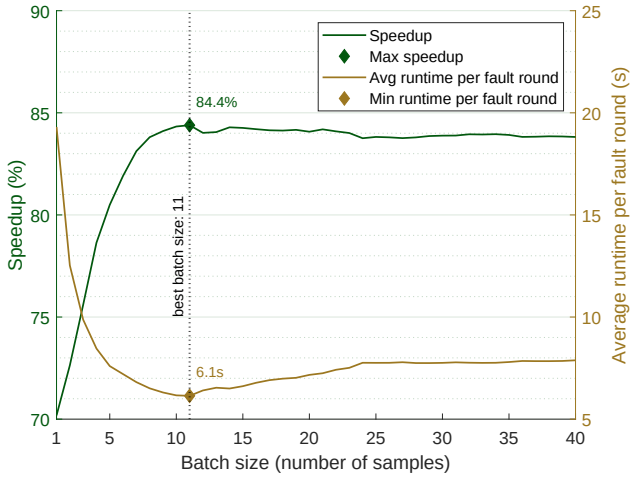
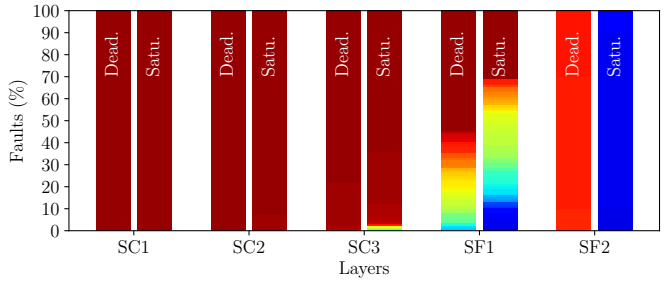
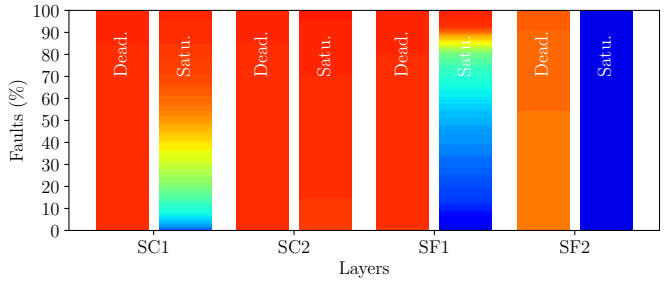


Fig. 10: Speedup using batched inference.



(a) N-MNIST SNN.



(b) IBM DVS128 Gesture SNN.

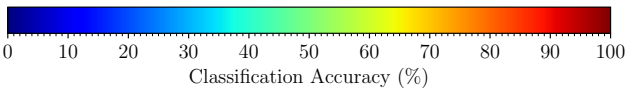


Fig. 11: Resiliency analysis for dead and saturated neuron faults.

for many input samples simultaneously and, on the other hand, by increasing the batch size we reduce the data transfer time to and from the GPU. However, at some specific batch size, the bottleneck of the GPU communication is reached, saturating the speedup improvement.

C. Demonstrations

1) Neuron hard faults: In the first experiment, we inject single hard neuron faults considering all neurons. We ran two separate FI campaigns for dead and saturated neuron faults.

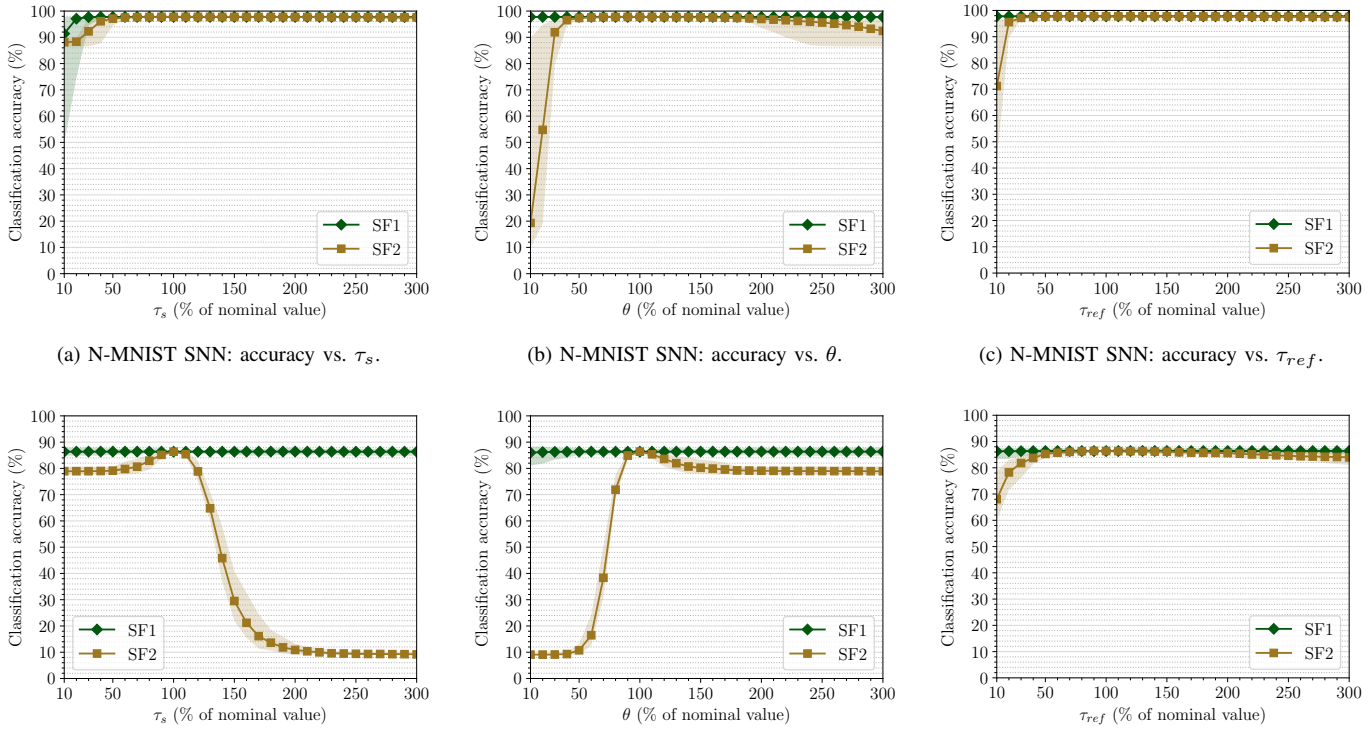
Fig. 11 shows a possible visualization of the results using comparative bar plots. The x-axis shows the different layers of the network and for each layer there are two bars, one for the dead and one for the saturated neuron faults. Note that pooling layers were excluded from the analysis since their functionality is to aggregate regions of spikes of their previous layers and do not contain any spiking neurons. A bar is separated into chunks of different colors, each corresponding to a specific classification accuracy according to the color shading shown at the bottom of Fig. 11. The height of the chunk projected on the y-axis shows the percentage of neurons in this layer which when exhibit this type of fault the classification accuracy drops to the value indicated by the color of the chunk.

A first observation from Fig. 11 is that saturated faults have a far stronger impact on the classification accuracy compared to dead faults. At the output layer, a saturated neuron always wins the race, thus samples from all classes except the one corresponding to the winning neuron are always misclassified. In the case of a dead neuron, an input with class label corresponding to this neuron is always mislassified, while samples from other classes are not affected. Taking the N-MINST SNN as an example, a saturated neuron at the output layer causes the accuracy to plummet to a value of 10% on average, while a dead neuron reduces the accuracy by 10% on average. The fact that saturated faults are more lethal than dead faults is also evident in the SF1 layer of the two networks. The IBM DVS128 Gesture SNN is impacted also in the SC1 layer, while the N-MNIST SNN is insensitive to faults in the first two layers and in the third layer only a 2% of neurons are critical.

2) Neuron parametric faults: The effect of neuron parametric faults on the classification accuracy is shown in Fig. 12. For a given layer, we vary the τ_s, τ_{ref}, and θ parameters in the SRM for one neuron at a time. The main curve represents in the y-axis the average classification accuracy observed across all neurons of the layer as a function of the parameter deviation in the x-axis expressed in % of the nominal value, i.e., 100% corresponds to zero deviation. The colored region surrounding the curve demonstrates the minimum and maximum classification accuracy. Neuron parametric faults were found to have a noticeable impact only for the output layer, thus in Fig. 12 we show only the results for the output and last hidden layer.

The N-MNIST SNN shows a high resilience even at the output layer. The classification accuracy starts degrading when τ_s, θ, and τ_{ref} are reduced at 40%, 40%, and 20%, respectively, while positive deviations have practically no effect as accuracy degradation starts being noticeable only for θ when it increases beyond 200%. In contrast, the SNN DVS128 Gesture SNN shows vulnerability even for small τ_s and θ fluctuations, while it shows a high degree of resiliency for τ_{ref}.

As mentioned in Section III-A2, increasing τ_s, decreasing θ, or decreasing τ_{ref} makes the neuron spike more easily. At the extreme, this direction of deviation may make the neuron saturate, which, as we observed in Section V-C1, is far more fatal than a dead fault. This behavior can be observed in Fig. 12. Extreme positive deviation of τ_s for the IBM DVS128 Gesture SNN has an effect equivalent to neuron saturation as the accuracy drops to 9.09%, i.e., only 1 out of 11 classes is predicted correctly. Similarly, extreme negative deviation of θ is equivalent to neuron saturation in both networks.



(a) N-MNIST SNN: accuracy vs. τ_s. (b) N-MNIST SNN: accuracy vs. θ. (c) N-MNIST SNN: accuracy vs. τ_{ref}.

(d) IBM DVS128 Gesture SNN: accuracy vs. τ_s. (e) IBM DVS128 Gesture SNN: accuracy vs. θ. (f) IBM DVS128 Gesture SNN: accuracy vs. τ_{ref}.

Fig. 12: Resiliency analysis for neuron parametric faults.

3) Synapse faults: We consider four different synapse faults, namely dead, negatively and positively saturated, and bit-flipped synapses. The negative and positive saturation values were set to -10 and $+10$, respectively, which are extreme considering the synaptic weight distribution after training. For the bit-flipped synapse faults we consider that the hardware accelerator uses an 8-bit integer data format. For each synapse fault model, we performed an exhaustive FI campaign with single synapse faults covering all synapses connecting the last hidden layer to the output layer. The result is illustrated in the form of heat maps in Figs. 13 and 14. Each square in the heat map corresponds to a unique synaptic connection and the square's color represents the network's resultant classification accuracy when the synapse fault is introduced. The framework outputs these heat maps such that the pre-synaptic neurons are placed in the x-axis and the post-synaptic neurons are placed in the y-axis. As the number of neurons can be very high, the framework offers the possibility to re-shape the area of the heat map for illustration purposes.

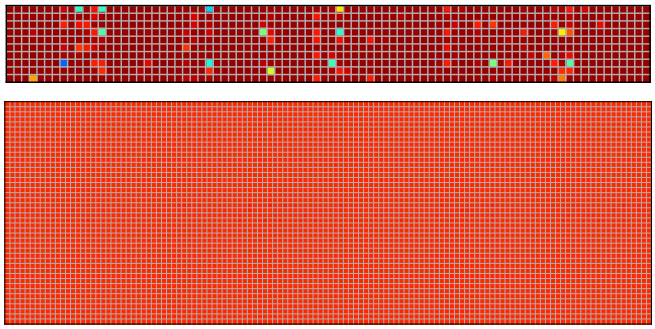
From Figs. 13a and 13b, we observe that only a few synapses can affect the classification accuracy if they become dead or negatively saturated, and the drop in the classification accuracy is in most cases small. A dead synapse fault zeros the spikes passing from the synapse, while a negative saturated synapse fault converts the spikes to large negative spikes. In both cases, the membrane potential of the post-synaptic neuron reduces and at the extreme the neuron never fires behaving like a dead neuron. In contrast, positive saturated synapse faults can have a greater impact, as shown in Fig. 13c. This is because they increase the spikes' strength to the point where the post-synaptic neuron fires easily, behaving at the extreme like a saturated neuron.

TABLE II: Statistics of the FI campaigns in Sections V-C1, V-C2 and V-C3.

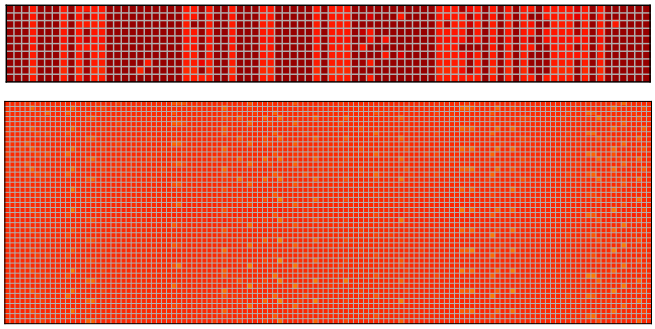
Network	Fault type	Fault rounds (#)	Total runtime (sec)	Runtime per fault (sec)
N-MNIST	Neuron hard	13036	163982	12.58
	Neuron parametric	8460	32290	3.82
	Synapse	9240	15404	1.67
IBM DVS128 Gesture	Neuron hard	50198	66989	1.33
	Neuron parametric	47070	18066	0.34
	Synapse	61952	4768	0.08

From Fig. 14, we observe that the effect on the classification accuracy increases with the bit position, with the most significant bits (MSBs) being the most critical. We observe also that the N-MNIST SNN is the most vulnerable to this synapse fault type, as even the least significant bit (LSB) is critical for many synapses.

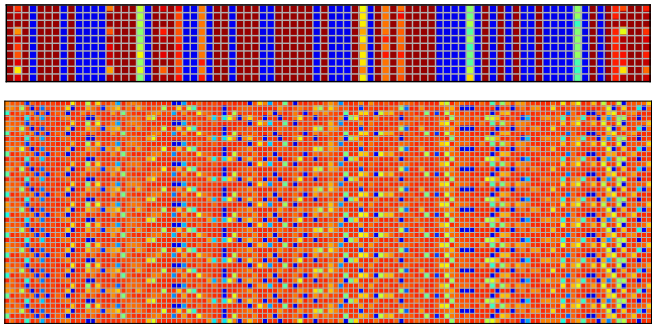
4) Runtime: Table II summarizes the statistics of the FI campaigns in the above Sections V-C1, V-C2 and V-C3. It shows the total number of fault rounds, total runtime, and average runtime per fault round per network and per fault type. Overall, these experiments involved 189956 fault rounds and took around 3.5 days to complete. Synapse fault simulation presents the smallest runtime, while for neuron parametric faults the runtime is one order of magnitude smaller than neuron hard faults.



(a) Dead synapse fault.



(b) Negatively saturated synapse fault.



(c) Positively saturated synapse fault.

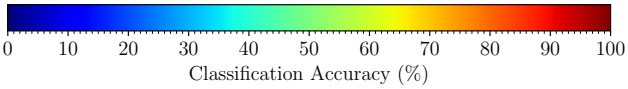


Fig. 13: Resiliency analysis for synapse faults in synapses connecting the last two layers. In each sub-figure the result for the N-MNIST SNN is shown at the top and for the IBM DVS128 Gesture SNN at the bottom.

5) Training in the presence of faults: Herein, *SpikeFI* is used to assess the ability of a SNN to learn in the presence of faults. The FI experiment consists of several cumulative fault rounds, where in each fault round 10 new faults are added with respect to the previous one. The fault sites within a fault round are randomly assigned across the network excluding the output layer. The fault target can be any neuron or synapse, and a fault function from those defined in the fault model library is randomly assigned. For every fault round, the faults are injected into the network and then training is performed. Fig. 15 shows the learning curves for the N-MNIST SNN. Each learning curve corresponds to a separate fault round and results in a new instance of the SNN. As it can be seen, the SNN is capable of learning around and withstanding multiple faults. The nominal fault-free accuracy is reached for up to 100 faults, although the learning rate slows down as the number of faults increases. The classification accuracy starts dropping after 100 simultaneous faults occur in the network, with the drop increasing with the number of faults. This experiment shows that SNN hardware accelerators used for training present a high degree of passive tolerance to faults existing prior to training, and that re-training when faults occur is a workable active fault tolerance approach at the expense of bringing the network temporarily offline.

VI. Conclusions

We described *SpikeFI*, an open-source GPU-accelerated FI framework for SNNs. *SpikeFI* is built on top of the popular SLAYER framework used for training SNNs. It has built-in a library of mainstream neuron and synapse fault types modeled onto the SRM that can be extended and customized by the user if desired. *SpikeFI* enables highly flexible FI experiments for any arbitrary SNN model. Each FI experiment is composed of independent fault rounds, where, in turn, each fault round can be composed of single or multiple faults of different types. The fault duration can be permanent or transient and the fault site can be specified or can be randomly assigned layer-wise or network-wise. *SpikeFI* can be used to train a SNN with faults to make it robust to the presence of faults, study the effect on the training accuracy for faults manifesting during a long training process, or study the effect on the inference accuracy for faults occurring post-training so as to pinpoint critical faults and develop smart and low-cost test and fault tolerance techniques. *SpikeFI* features various speed-up optimization tricks and various results visualization functions compatible with all fault types. *SpikeFI* was demonstrated on two SNNs designed for the N-MNIST and IBM DVS128 Gesture datasets, which are the two most common benchmark datasets within the neuromorphic community.

Acknowledgments

This work was funded by the ANR RE-TRUSTING project under Grant N^{o} ANR-21-CE24-0015-03 and by the European Network of Excellence dAIEDGE under Grant Agreement N^{o} 101120726. The work of T. Spyrou was supported by the Sorbonne Center for Artificial Intelligence (SCAI) through Fellowship.

References

[1] K. Roy, A. Jaiswal, and P. Panda, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, no. 7784, pp. 607–617, Nov. 2019.

[2] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nat. Comput. Sci.*, vol. 2, no. 1, pp. 10–19, Jan. 2022.

[3] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker Project," *Proc. IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[4] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

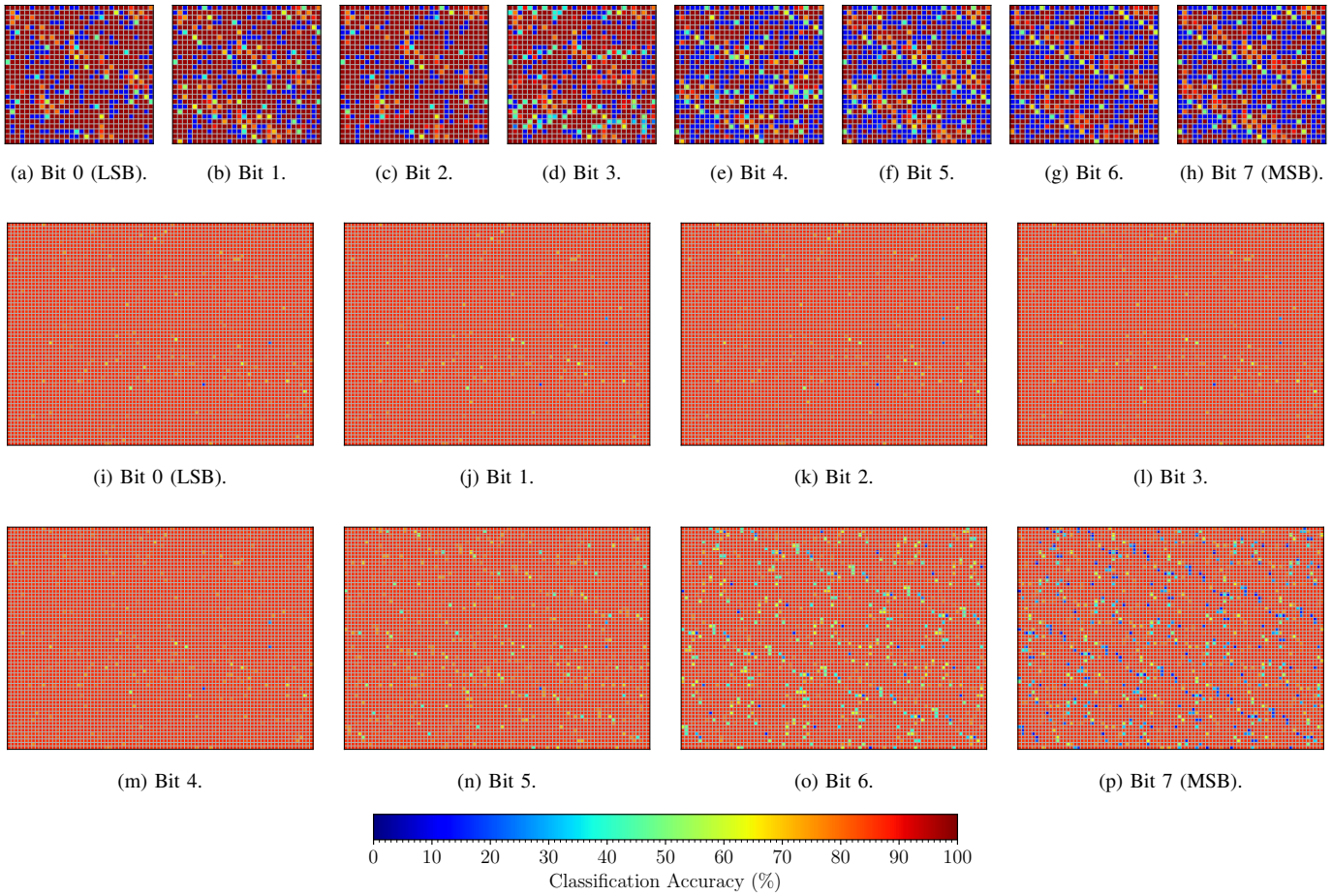


Fig. 14: Resiliency analysis for bit-flips in synapses connecting the last two layers for the N-MNIST SNN (top heat maps) and IBM DVS128 Gesture SNN (bottom heat maps).

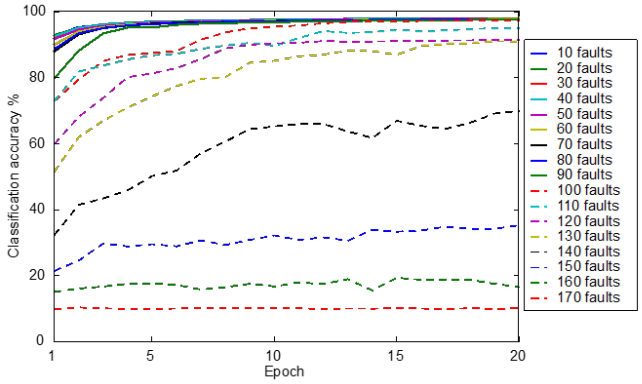


Fig. 15: Learning curves of the training of the N-MNIST SNN in the presence of multiple faults of various types.

[5] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018.

[6] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May/Jun. 2010, pp. 1947–1950.

[7] B. V. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proc. IEEE*, vol. 102, no. 5, pp. 699–716, Apr. 2014.

[8] H.-Y. Tseng, I-W. Chiu, M.-T. Wu, and J. C.-M. Li, "Machine learning-based test pattern generation for neuromorphic chips," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2021.

[9] I-W. Chiu, X.-P. Chen, J. S.-I. Hu, and C.-M. J. Li, "Automatic test configuration and pattern generation (atcpg) for neuromorphic chips," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Oct./Nov. 2022.

[10] S. A. El-Sayed, T. Spyrou, L. A. Camuñas-Mesa, and H.-G. Stratigopoulos, "Compact functional testing for neuromorphic computing circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 7, pp. 2391–2403, Jul. 2023.

[11] T. Spyrou and H.-G. Stratigopoulos, "On-line testing of neuromorphic hardware," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2023.

[12] C. D. Schuman *et al.*, "Resilience and robustness of spiking neural networks for neuromorphic systems," in *Proc. Int. Jt. Conf. Neural Netw. (IJCNN)*, Jul. 2020.

[13] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Automatic abstraction and fault tolerance in cortical microachitectures," in *Proc. ACM/IEEE Annual Int. Symp. Comput. Archit. (ISCA)*, Jun. 2011, pp. 1–10.

[14] S. Karim *et al.*, "Assessing self-repair on FPGAs with biologically realistic astrocyte-neuron networks," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2017, pp. 421–426.

[15] A. P. Johnson *et al.*, "Homeostatic fault tolerance in spiking neural networks: A dynamic hardware perspective," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 2, pp. 687–699, 2018.

[16] J. Liu, J. Harkin, L. P. Maguire, L. J. McDaid, and J. J. Wade, "SPANNER: A self-repairing spiking neural network hardware architecture," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 4, pp. 1287–1300, Apr. 2018.

[17] S. A. El-Sayed, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Self-testing analog spiking neuron circuit," in *Proc. Int. Conf. Synth. Model. Anal. Simulat. Methods Appl. Circuit Design (SMACD)*, Jul. 2019.

[18] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Neuron fault tolerance in spiking neural networks," in *Proc. Design Autom. Test Europe Conf. (DATE)*, Feb. 2021, pp. 743–748.

[19] R. V. W. Putra, M. A. Hanif, and M. Shafique, "SoftSNN: Low-cost fault tolerance for spiking neural network accelerators under soft errors," in *Proc. 59th Design Autom. Conf. (DAC)*, Jul. 2022, p. 151–156.

[20] A. Saha, C. Amarnath, and A. Chatterjee, "A resilience framework for synapse weight errors and firing threshold perturbations in RRAM spiking neural networks," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2023.

[21] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Reliability analysis of a spiking neural network hardware accelerator," in *Proc. Design Autom. Test Europe Conf. (DATE)*, Mar. 2022, pp. 370–375.

[22] E. Vatajelu, G. Di Natale, and L. Anghel, "Special session: Reliability of hardware-implemented spiking neural networks (SNN)," in *Proc. IEEE VLSI Test Symp. (VTS)*, Apr. 2019.

[23] A. B. Gogebakan, E. Magliano, A. Carpegna, A. Ruospo, A. Savino, and S. Di Carlo, "SpikingJET: Enhancing fault injection for fully and convolutional spiking neural networks," in *Proc. IEEE Int. Symp. On-Line Test. Robust Syst. Des. (IOLTS)*, Jul. 2024.

[24] J. K. Eshraghian *et al.*, "Training spiking neural networks using lessons from deep learning," *Proc. IEEE*, vol. 111, no. 9, pp. 1016–1054, Sep. 2023.

[25] S. B. Shrestha and G. Orchard, "SLAYER: Spike layer error reassignment in time," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2018, pp. 1412–1421.

[26] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2017.

[27] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018.

[28] A. Mahmoud *et al.*, "PyTorchFI: A runtime perturbation tool for DNNs," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun./Jul. 2020, pp. 25–31.

[29] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A flexible fault injection framework for tensorflow applications," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2020, pp. 426–435.

[30] N. Narayanan, Z. Chen, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Fault injection for TensorFlow applications," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 2677–2695, Jul./Aug. 2023.

[31] Y. He, P. Balaprakash, and Y. Li, "FIdelity: Efficient resilience analysis framework for deep learning accelerators," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2020, pp. 270–281.

[32] C. Bolchini, L. Cassano, A. Miele, and A. Toschi, "Fast and accurate error simulation for CNNs against soft errors," *IEEE Trans. Comput.*, vol. 72, no. 4, pp. 984–997, Apr. 2023.

[33] L. M. Luza *et al.*, "Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 4, pp. 1867–1882, Oct./Dec. 2022.

[34] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "*BinFI*: An efficient fault injector for safety-critical machine learning systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2019.

[35] A. Chaudhuri, J. Talukdar, F. Su, and K. Chakrabarty, "Functional criticality analysis of structural faults in AI accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 12, pp. 5657–5670, Dec. 2022.

[36] A. Gavarini, A. Ruospo, and E. Sanchez, "SCI-FI: a smart, accurate and unintrusive fault-injector for deep neural networks," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2023.

[37] A. Ruospo *et al.*, "Assessing convolutional neural networks reliability through statistical fault injections," in *Proc. Design Autom. Test Europe Conf. (DATE)*, Apr. 2023.

[38] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2017, pp. 249–258.

[39] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: Dynamic fault injection for GPUs," in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2021, pp. 284–291.

[40] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives," *IEEE Des. Test*, vol. 40, no. 2, pp. 8–58, Apr. 2023.

[41] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, Feb. 2023.

[42] H. Stratigopoulos, T. Spyrou, and S. Raptis, "Testing and reliability of spiking neural networks: A review of the state-of-the-art," in *Proc. IEEE Int. Symp. Defect Fault Toler. VLSI Nanotechnol. Syst. (DFT)*, Oct. 2023.

[43] M. H. Ahmadilivani, M. Taheri, J. Raik, M. Daneshtalab, and M. Jenihhin, "A systematic literature review on hardware reliability assessment methods for deep neural networks," *ACM Comput. Surv.*, vol. 56, no. 6, Jan. 2024.

[44] P. Rech, "Artificial neural networks for space and safety-critical applications: Reliability issues and potential solutions," *IEEE Trans. Nucl. Sci.*, vol. 71, no. 4, pp. 377–404, Jan. 2024.

[45] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Netw.*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[46] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition*, Cambridge University Press, 2014.

[47] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Netw.*, vol. 111, pp. 47–63, Mar. 2019.

[48] S. A. El-Sayed, T. Spyrou, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Spiking neuron hardware-level fault modeling," in *Proc. 26th IEEE Int. Symp. On-Line Test. Robust Syst. Des. (IOLTS)*, Jul. 2020.

[49] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., pp. 8024–8035. Curran Associates, Inc., 2019.

[50] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Front. Neurosci.*, vol. 9, Nov. 2015, Article 437.

[51] A. Amir *et al.*, "A low power, fully event-based gesture recognition system," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017.

[52] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.