POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

TRAINING-TIME FAULT INJECTOR FOR SPIKING NEURAL NETWORKS

Supervisors

Prof. STEFANO DI CARLO

Prof. ALESSANDRO SAVINO

MSc. ENRICO MAGLIANO

MSc. ALESSIO CAVIGLIA

Candidate

MERIC ULUCAY

DEC 2025

Summary

Spiking Neural Networks (SNNs) have gained significant attention as energy efficient computational models capable of processing information through sparse, event-driven signaling. Their temporal processing abilities and compatibility with neuromorphic hardware make them suitable for applications that require fast response and low power consumption, including robotics, biomedical monitoring, edge intelligence, and space systems. Despite these advantages, real hardware deployments expose SNNs to numerous reliability risks arising from manufacturing imperfections, thermal variability, radiation exposure, or device aging. These physical factors can corrupt internal variables or permanently damage computational elements. Although SNNs are often assumed to be resilient due to their biological inspiration, the degree to which they tolerate hardware faults during training remains largely unexplored. Understanding how such faults influence both the learning dynamics and final model performance is essential for the development of reliable neuromorphic systems.

This thesis provides a systematic investigation of the robustness of SNNs when hardware-inspired faults are introduced during training. Previous research has focused mostly on injecting faults during inference. In contrast, this study examines how faults interact with the learning process itself. Two common error sources from neuromorphic hardware are modeled. Bit-flip faults represent transient disruptions caused by phenomena such as radiation-induced single-event upsets. Stuck-at faults represent permanent failures that may occur due to aging or physical degradation. Unlike studies that consider only weight corruption, this work injects faults into a wide variety of SNN components. These include synaptic weights, gradients, input spike tensors, hidden activations, membrane potentials, membrane decay constants (β), and firing thresholds. By including both learnable parameters and internal state variables, the framework captures a realistic spectrum of error behaviors that may arise in physical SNN hardware.

A training-time fault injection framework was developed to support this analysis. The injector is integrated directly into the PyTorch training loop and operates at the level of individual tensor elements, enabling localized and realistic error modeling. At each training iteration, the framework receives a list of all eligible

parameters and state variables, including weights, gradients, input spikes, hidden activations, membrane potentials, β values, and firing thresholds. One element is then selected uniformly at random, reflecting the assumption that hardware faults occur independently of software-defined structure. Once a target is chosen, either a bit-flip or stuck-at operation is applied. Bit-flip faults toggle a specific bit within the numerical representation of the value, simulating transient soft errors. Stuck-at faults overwrite the entire value with either an all-zero or all-one pattern, emulating permanent hardware failures caused by aging or physical degradation.

The injector is capable of modeling both temporary and long-lasting disruptions. Transient faults, introduced via bit-flips, typically affect the tensor only momentarily and may be overwritten by subsequent computations or parameter updates. Permanent faults, implemented through stuck-at assignments, persist for the remainder of training unless manually reinitialized. During training, each fault event is logged along with its type, affected parameter, bit position, layer index, and the value before and after corruption. At the end of every epoch, the validation accuracy of the faulty model is recorded and compared with that of an identical fault-free baseline. The difference between these two values forms the basis of the impact measurement used throughout the analysis. By aggregating these differences across multiple training runs and fault occurrences, the study evaluates average fault impact as a function of fault type, bit position, layer depth, and injection epoch. Although faults generally reduce accuracy, occasional positive changes were observed, possibly because small disturbances can sometimes counteract overfitting and act as unintended regularizers.

Experiments are carried out on three benchmark datasets that represent visual, auditory, and temporal neuromorphic tasks. These datasets are NMNIST, AudioMNIST, and the Spiking Heidelberg Digits (SHD). For each dataset, a fully connected SNN model is trained using surrogate gradient descent, and the proposed injector is applied throughout training. The results reveal several consistent sensitivity patterns across the models. One important observation is related to the timing of the fault. Faults that occur at later stages of training cause significantly larger accuracy degradation than those that occur early. Early faults are often overwritten or compensated for by subsequent parameter updates. However, once the model has stabilized, faults become more persistent and influence the final decision boundaries more strongly.

Another prominent pattern involves the sensitivity of different parameter types. Faults in weights, β values, and firing thresholds consistently produce the largest accuracy drops. These components strongly influence neuron dynamics and therefore represent critical points of failure. On the other hand, faults in activations, membrane potentials, or input events tend to have smaller effects because these variables are temporary and recalculated frequently. The analysis also shows a clear relationship between fault impact and layer depth in most experiments. Faults

in later layers often result in larger deviations than faults in earlier layers. The effect is particularly visible in the AudioMNIST and SHD experiments and reflects the fact that disruptions occurring closer to the output stage have a more direct influence on classification.

Despite these common trends, the results also reveal significant variations between the models. The NMNIST network shows strong resilience to both transient and permanent faults. Most injected faults produce almost no measurable degradation in this model. In contrast, the AudioMNIST and SHD networks exhibit much clearer patterns of vulnerability. Their accuracy reductions reach values around one to one and a half percentage points in the stuck-at experiments, and the differences between fault types and layers become more pronounced. These findings demonstrate that SNN robustness is strongly dependent on the dataset and architecture. Therefore, conclusions derived from a single configuration may not generalize across all neuromorphic workloads. Beyond the empirical results, this thesis provides a training-time fault injection framework that enables controlled examination of how SNNs respond to disruptions while learning. Because faults are introduced directly into the optimization process, the tool captures interactions between learning dynamics and hardware-inspired errors that inference-only analyses cannot reveal. This makes the injector not only a diagnostic tool for understanding fault behavior but also a potential component of robustness-oriented training strategies.

Future work may extend the experimental scope to broader datasets and additional SNN architectures, allowing a deeper assessment of how data complexity and network structure influence fault sensitivity. The injector could also be integrated into training pipelines that intentionally introduce noise or faults to improve robustness. Finally, applying the approach to hardware-in-the-loop experiments would help validate the findings on real neuromorphic devices and support the development of standardized benchmarks for resilient SNN design.

Acknowledgements

I would like to express my deepest gratitude to Prof. Stefano Di Carlo, who introduced me to this topic and guided me throughout the development of this thesis. I am equally grateful to my supervisors, Enrico Magliano and Alessio Caviglia, for their constant support, patience, and clarity whenever I found myself confused or uncertain. I would also like to thank Alessio Carpegna, whose carefully drawn explanations helped me understand several concepts.

My heartfelt thanks go to Anil Bayram Gogebakan, who encouraged me through every setback and whose ideas and presence consistently helped me move forward. I am profoundly grateful to my parents, Sevim Ulucay and Unal Ulucay, who, despite being 3000 kilometers away, made me feel their support every single day.

I would also like to thank Selen Karakas, who stood by me through every emotional high and low; Goktug Sami Yavuz, whose midday coffee breaks made long days more bearable; Federica Schena and Benedetto Spadavecchia, who gave me the best escape I could have asked for in Bari; and Deniz Kasap, who supported me wholeheartedly and followed my thesis progress so closely despite the distance. Special thanks go to Oğuzhan Akgün, the most valuable member of our asırı gizli group; Ecem Cosan and Ozge Dalaklı, for making Friday evenings something to look forward to; Koray Oskay, the undefeated champion of our Monopoly nights; and Eren Olmez, who always arrived with the most entertaining and perfectly timed gossip updates.

Finally, I wish to thank the entire SMILIES Lab for brightening my weeks with beach volley.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

SNN
Spiking Neural Network

STDP
Spike-Timing-Dependent Plasticity

LIF
Leaky Integrate-and-Fire

ALIF
Adaptive LIF

AdEx
Adaptive Exponential Integrate-and-Fire

FC
Fully Connected

MSE
Mean Squared Error

ANN
Artificial Neural Networks

SFI
Statistical Fault Injection

SHD

Spiking Heidelberg Digits

DVS

Dynamic Vision Sensor

ATIS

Asynchronous Time based Image Sensor

SCNN

Spiking Convolutional Neural Network

CNN

Convolution Neural Network

Chapter 1

Introduction

SNNs have gained increasing attention as a promising alternative to conventional neural models due to their event driven computation, temporal encoding capabilities, and compatibility with neuromorphic hardware. By processing information through sparse spike based dynamics rather than continuous activations, SNNs enable low power and real time computation suitable for embedded and edge level platforms. These properties make SNNs strong candidates for applications requiring fast sensory response, autonomous processing, and strict energy constraints, including brain-inspired robotics, biomedical signal interpretation, and low power edge intelligence. In addition to their efficiency, the temporal and biologically inspired nature of SNN offers advantages in adaptability and functional robustness, particularly when deployed on unreliable or noise prone hardware. Such characteristics have recently motivated interest in SNNs for mission critical scenarios that demand both energy efficiency and resilience.

Although SNNs share common computational principles, their resilience to faults is far from uniform. Differences in architecture depth, parameter distribution, neuron model configuration, and dataset characteristics can lead to markedly different fault sensitivities. As a result, understanding fault behavior requires not only analyzing injected faults in isolation, but also examining how these architectural and dataset-specific factors influence the way errors propagate through the network. This motivates a systematic evaluation across multiple SNN models trained on diverse datasets, allowing a more complete characterization of the conditions under which SNNs become vulnerable or remain robust.

A prominent example of such scenarios is space and aerospace systems, where electronic components are continuously exposed to radiation, thermal variability, and aging effects that induce hardware level faults. These reliability hazards directly affect autonomous onboard systems used in satellite navigation, planetary exploration, and deep space missions, where hardware interventions are impossible and energy budgets are severely limited. While SNNs present a compelling

computational model for these conditions, their resilience under hardware faults remains insufficiently characterized. Consequently, assessing how SNNs behave when internal parameters are corrupted is crucial before these models can be safely deployed in fault prone environments.

Although SNNs are often assumed to inherit the robustness of biological neural systems, recent empirical studies indicate that hardware level faults can significantly degrade classification accuracy, disrupt spike timing, or induce persistent malfunctioning neurons, particularly when injected after the training phase. Existing research primarily analyzes fault impact at the inference stage, overlooking the network's potential to adapt to faulty conditions during learning. Since learning mechanisms such as surrogate gradient based updates or STDP modify neuron and synapse behavior over time, injecting faults during training may allow the model to compensate for malfunctioning components, thereby improving post-deployment resilience [1][2]. However, the extent to which training time faults enhance or degrade final model robustness remains insufficiently explored.

This thesis addresses this gap by investigating the impact of hardware induced faults introduced during the training phase of SNNs, addressing the limited focus in prior work on post training fault analysis. Faults are modeled at the bit level and encompass both transient perturbations and permanent defects, allowing the study to approximate realistic error mechanisms such as radiation induced flips or long term device degradation. Instead of restricting fault modeling to weights alone, the proposed framework injects faults into a broad set of learnable and run time elements that captures how diverse fault locations influence learning trajectories. By introducing faults while the network is actively adjusting its parameters through surrogate gradient learning, the study examines whether SNNs can maintain functional performance in the presence of corrupted components. The methodology is evaluated across three benchmark datasets which are NMNIST, AudioMNIST, and SHD that covers visual, auditory, and temporal neuromorphic tasks to assess how fault sensitivity varies across modalities. Overall, this study provides a unified perspective on fault behavior during learning, highlights the differing effects of transient and permanent faults, and offers insights relevant to deploying low power neuromorphic systems in fault prone environments.

The remainder of this thesis is organized as follows. Chapter 2 presents the necessary background on Spiking Neural Networks, fault models, and the datasets used throughout this work. Chapter 3 introduces the methodological framework, including data preprocessing pipelines, model architectures, and the proposed fault injection strategy applied during training. Chapter 4 will present the experimental setup, evaluation metrics, and observed results across different fault scenarios and datasets. Finally, Chapter 5 will summarize the findings, and outline potential directions for future research on resilient neuromorphic learning systems.

Chapter 2

Background Research

2.1 Spiking Neural Network (SNN)

SNNs represent the third generation of artificial neural networks, inspired by the information processing principles of biological neural systems. Unlike conventional artificial or deep neural networks, which operate on continuous real valued activations, SNNs communicate using discrete events known as spikes. Each spike corresponds to a binary and temporally localized signal, allowing information to be represented not only by the presence or absence of a spike but also by its precise timing [3]. This temporal dimension introduces richer dynamics and enables the processing of event based or time varying data such as sound, vision, or sensory signals in neuromorphic systems.

From a computational standpoint, SNNs incorporate time as an explicit variable into the neural processing model. Each neuron maintains a membrane potential that evolves according to the timing and polarity of incoming spikes, the synaptic strength of its connections, and its intrinsic parameters such as membrane decay and firing threshold. When this potential exceeds a threshold, the neuron emits a spike and resets, propagating activity to downstream neurons. This event driven mechanism makes computation inherently sparse and asynchronous, as operations occur only when spikes are present, rather than at every simulation step [3] [4] .

Compared to deep neural networks, which require continuous and dense matrix multiplications, SNNs exhibit remarkable energy efficiency and biological plausibility. The discrete, event driven nature of spike processing reduces redundant operations and enables in memory computation, where memory and processing occur locally within the same neuron-synapse structure. Such characteristics make SNNs particularly attractive for neuromorphic hardware implementations [5]. These platforms exploit the sparse, temporal, and parallel properties of SNNs to deliver high performance at ultra-low power, ideal for embedded or edge AI applications

where energy and latency constraints dominate.

Beyond hardware efficiency, SNNs also introduce a new representational paradigm which is computation through spike timing and temporal codes. Instead of static feature activations, information is carried by spatiotemporal spike patterns, which can naturally encode dynamic sensory data and support temporal reasoning. This capability allows SNNs to bridge the gap between neuroscience and machine learning, offering models that are not only computationally efficient but also closer to the principles of biological learning and perception.

However, training SNNs remains a major challenge due to the non-differentiable nature of spike generation. The discontinuous spiking function prevents direct use of traditional gradient based learning methods such as backpropagation. To address this, several learning paradigms have emerged. This paradigms are ranging from biologically motivated local rules such as STDP to differentiable surrogate gradient techniques that approximate spike derivatives. These developments have gradually improved the trainability of SNNs and opened new research directions in temporal pattern recognition, sensor fusion, and fault-resilient computing [3, 5].

In summary, SNNs combine temporal computation, event-driven sparsity, and neuromorphic efficiency within a unified framework. They serve as a promising computational paradigm that not only enhances energy efficiency but also deepens the connection between artificial and biological intelligence. It is bridging theoretical neuroscience, algorithmic design, and hardware realization [3, 5].

2.1.1 Biological Inspiration

The architecture of SNNs is directly inspired by the human brain's neuronal communication mechanism. Neurons communicate via discrete electrical impulses that are called spikes [3]. Each neuron maintains a membrane potential that integrates incoming synaptic currents over time. When the potential exceeds a specific threshold, the neuron fires a spike, transmitting it to connected neurons via synapses. Then, the potential resets, and the neuron enters a brief refractory period before it can fire again [6]. This process mirrors biological signal transmission observed in cortical circuits, where neurons encode sensory information through spike timing rather than continuous voltage levels. The explicit modeling of time in SNNs allows for dynamic behavior such as adaptation, inhibition, and synchronization phenomena that are difficult to reproduce in conventional networks [6].

2.1.2 Neuron Types

Spiking neurons form the core computational units of SNNs. Unlike traditional artificial neurons that process static and continuous signals, spiking neurons incorporate temporal dynamics, integrating incoming spikes over time and emitting an output spike once a firing threshold is reached. This mechanism allows SNNs to emulate the event driven computation observed in biological nervous systems [3, 5].

A commonly used spiking neuron model is the LIF neuron. The LIF neuron models the membrane potential as a leaky capacitor that accumulates input currents and gradually decays toward a resting value when no input is present. When the potential exceeds a threshold, a spike is emitted and the neuron resets its state. The membrane dynamics can be described by the differential equation:

$$\tau_m \frac{dV_m(t)}{dt} = -\left(V_m(t) - V_{\text{rest}}\right) + R_m I(t), \tag{2.1}$$

where $V_m(t)$ is the membrane potential, $I(t)$ is the input current, R_m the membrane resistance, and T_m the membrane time constant.

This process is illustrated in Figure 2.1, where the top panel shows input spikes, the middle panel shows membrane potential evolution, and the bottom panel displays the resulting output spikes once the threshold is reached.

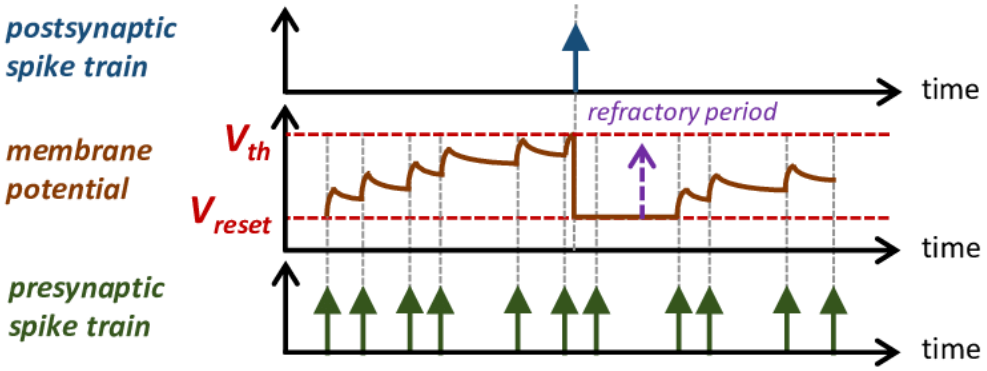

Figure 2.1: Illustration of Lapicque's LIF neuron model showing the relationship between input spikes, membrane potential dynamics, and output spikes [7].

To make simulation feasible in digital environments, continuous membrane dynamics are often discretized over time. The discrete-time update of the membrane potential at time step n can be expressed as:

$$V_m[n] = \begin{cases} \beta \cdot V_m[n-1] + W \cdot s_{in}[n], & \text{if } V_m[n-1] \leq V_{th} \\ V_m[n-1] - V_{th} + W \cdot s_{in}[n], & \text{if } V_m[n-1] > V_{th} \end{cases} \tag{2.2}$$

where $\beta = e^{-\Delta t/\tau_m}$ represents the leak factor, W denotes the synaptic weights, and $s_{in}[n]$ and $s_{out}[n]$ correspond to input and output spikes, respectively [6].

Several variants of the LIF neuron have been developed to capture additional biological behaviors. The ALIF model introduces an adaptation variable that increases the firing threshold after each spike, representing neuronal fatigue and promoting sparse activity [5]. Another biologically detailed model, the Izhikevich neuron, combines computational efficiency with the ability to reproduce complex firing patterns such as bursting, chattering, and spike-frequency adaptation [3]. The AdEx model further refines the dynamics by introducing an exponential term to more accurately represent membrane depolarization near firing threshold [5].

Together, these neuron models define how individual spiking units integrate inputs and generate discrete events, forming the microscopic foundation of all SNN architectures.

2.1.3 Layer Types

While neuron models define individual dynamics, layer types determine how large populations of neurons are connected and interact to process information. Each layer specifies the flow of spikes through the network from input encoding to high-level feature extraction and classification.

A conceptual overview of spike-based signal propagation within a layer is illustrated in Figure 2.2. Incoming spike trains $(X_{1,<t}, X_{2,<t}, \ldots, X_{n,<t}$ are weighted by synaptic connections $(W_{1,j}, W_{2,j}, \ldots, W_{n,j})$ and integrated by postsynaptic neurons to produce output spikes $(X_{j,t})$.

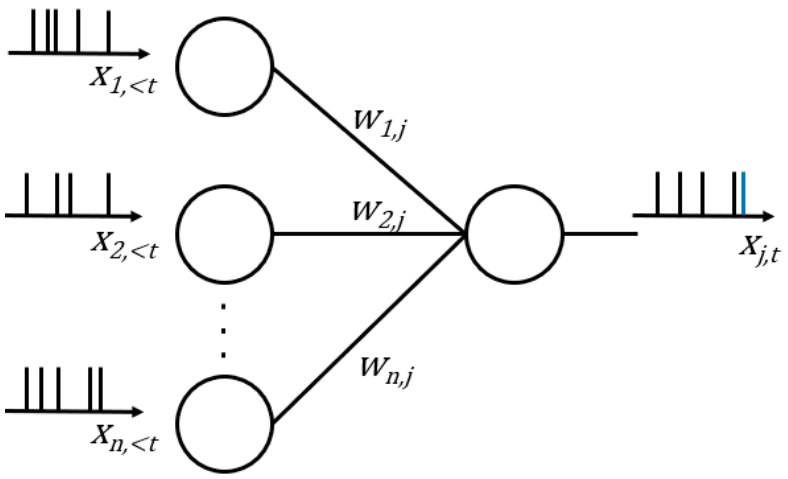

Figure 2.2: Schematic representation of spike-based computation within a neuron layer. Weighted input spikes are integrated to generate output spikes at time t [8].

The FC layer is the most basic structure in SNNs, where each neuron in one layer is connected to every neuron in the next. It enables global integration of

information but is computationally expensive for large-scale networks [5].

The Convolutional Layer introduces spatial locality by restricting connections to a local receptive field, similar to the operation of biological visual cortices. This makes SCNNs well suited for visual and auditory event based data, efficiently capturing spatiotemporal correlations through shared weights and sparse spike activations [3].

The Recurrent Layer extends temporal processing by allowing feedback connections, enabling neurons to retain memory of past inputs. This property enhances the ability of SNNs to process sequential data such as speech or event streams [5, 3].

Collectively, these layer types define how spiking neurons are arranged and interconnected to construct deep and functionally diverse SNN architectures capable of performing complex cognitive tasks.

2.1.4 Parameters of SNN

The behavior of a SNN is largely determined by several neuron level and layer level parameters that govern its temporal and dynamic responses. The membrane potential (V_m) represents the internal state of a neuron, integrating incoming spikes over time and determining when a spike is generated. A spike occurs once this potential surpasses the threshold (V_{th}), which defines the neuron's firing condition. A lower threshold increases sensitivity to inputs but may also lead to instability [6]. The membrane time constant (τ_m) regulates how quickly the potential decays toward its resting value, effectively determining the leakiness of the neuron, while the decay factor (β) serves as its discrete time approximation in simulation frameworks [4]. Information transfer between neurons is modulated by synaptic weights, which scale the influence of presynaptic activity. Together, these parameters shape how rapidly and sensitively neurons respond to stimuli, ultimately defining the overall temporal dynamics and computational properties of the SNN.

2.1.5 Learning Method

Learning in SNNs differs fundamentally from conventional ANNs, as information is encoded in discrete spike events over time rather than in continuous activations. Each neuron integrates incoming current into its membrane potential and emits a spike once this potential exceeds a firing threshold, thereby transmitting information through temporally precise events. However, this discrete and non-differentiable spiking process makes it difficult to apply standard gradient-based optimization methods directly.

To address this limitation, surrogate gradient methods are employed, which approximate the derivative of the spike activation function with a continuous

surrogate during backpropagation. This allows gradient-based learning in SNNs analogous to ANNs, enabling efficient supervised training using loss functions such as MSE or cross entropy. These methods make it feasible to train deep SNNs with modern optimizers like Adam while maintaining temporal dynamics [6].

Alternatively, unsupervised biologically inspired rules such as STDP can be used, where synaptic weights are adjusted based on the relative timing between pre- and post-synaptic spikes:

$$\Delta w = \begin{cases} A_+ e^{-\Delta t/\tau_+}, & \text{if } \Delta t > 0 \\ -A_- e^{\Delta t/\tau_-}, & \text{if } \Delta t < 0 \end{cases} \tag{2.3}$$

where $\Delta t = t_{post} - t_{pre}$ represents the temporal difference between spikes. In this work, supervised learning with surrogate gradient-based backpropagation is used, as it provides more stable convergence for large-scale datasets and facilitates consistent comparison across fault injection scenarios [3].

2.1.6 Python Interface

Python provides a practical environment for implementing spiking neural networks, offering high level tools that make it possible to translate theoretical SNN models into executable simulations. Deep learning libraries such as PyTorch supply the vectorized operations, automatic differentiation, and modular layer definitions needed to construct and train network architectures. On top of these foundations, specialized SNN oriented libraries extend PyTorch with components tailored to spiking dynamics.

Among these, snnTorch is widely used for prototyping because it introduces spiking neuron models such as the LIF unit directly as drop in layers. These modules maintain membrane potentials over time, apply decay, generate spikes when thresholds are crossed, and provide surrogate gradient approximations that enable gradient based learning. In this way, fully connected, convolutional, or recurrent architectures can be converted into SNN counterparts simply by replacing conventional activation functions with spiking neuron layers.

Overall, the Python ecosystem provides a flexible interface for SNN development: PyTorch defines the computational graph, SNN focused packages supply spiking dynamics, and domain specific libraries prepare data in time structured formats. This combination enables to prototype, analyze, and train SNN architectures efficiently while maintaining consistency with standard machine learning workflows.

2.2 Fault Injection

Fault injection is a controlled testing methodology used to evaluate the reliability and robustness of hardware and software systems by deliberately introducing faults

into their computational process [6]. The technique aims to emulate real world error conditions such as transient disturbances, permanent defects, or parametric deviations in order to observe how the system reacts, recovers, or loses functionality under fault [2]. Fault injection in SNNs enables researchers to simulate low level hardware faults within neuron and synapse models, allowing the systematic study of fault propagation, its impact on network behavior and performance [9]. Fault injection can be applied both in hardware, by physically disturbing the circuit, and in software, by simulating the effect of such faults within the model. Depending on the required accuracy and controllability, the injection can be performed through pure software simulation or hardware-based testing on physical devices.

2.2.1 Motivation

Fault injection studies are motivated by the increasing deployment of SNN based accelerators in energy and latency constrained edge devices and by the corresponding need to ensure dependability in safety critical applications [10]. Biological neural systems are often cited for their robustness. However, empirical fault injection experiments indicate that hardware implementations of both ANNs and SNNs can be vulnerable to hardware level faults, particularly when faults occur after model training and deployment [9]. This vulnerability arises because some faults can lead to critical functional changes that are not mitigated by the network's redundancy, resulting in silent data corruption or significant drops in inference accuracy [11]. Consequently, systematic fault injection is required both to identify the most critical fault types and locations and to quantify the limits of the network's natural resilience so that efficient mitigation strategies can be designed [1]. Performing such studies at transistor level is computationally prohibitive for large networks. Therefore, software-level fault injection is employed to emulate hardware faults.

2.2.2 Techniques

Fault injection can be performed at different abstraction levels, ranging from low-level hardware circuits to high-level software simulations. Depending on the purpose of the study, faults may target electrical signals, memory elements, or algorithmic variables that represent neuron or synapse behavior. Faults can be categorized in several complementary ways, such as by duration (transient or permanent), by functional impact (dead or saturated neurons, timing variations, or parametric drifts), or by their physical location in the system (weights, thresholds, or interconnects). [11]. Because these fault manifestations arise from different physical mechanisms and occur across different operational contexts, various methodological approaches have been developed to study them systematically.

From a methodological perspective, two main approaches exist. Hardware-based fault injection involves inducing real physical disturbances in neuromorphic devices like voltage or clock glitches, or radiation exposure. While these methods provide realistic insights, they are difficult to control and repeat. Software based or simulation based fault injection, on the other hand, emulates hardware defects in a modeled environment by manipulating internal variables such as weights, neuron thresholds, or membrane potentials. This approach is preferred for its scalability, repeatability, and precise control over injection timing and fault parameters. Instead of replicating every transistor level detail, behavioral fault models are derived from lower level analyses and implemented at the software level to mimic how physical faults would alter neuron and synapse dynamics [12, 9]. This enables large scale, systematic evaluation of SNN resilience while maintaining computational feasibility.

Because the total number of possible fault sites in a model can reach millions, testing every location exhaustively is impractical. To address this, SFI is used. In SFI, only a statistically representative subset of all possible faults is injected according to a controlled sampling plan. This allows researchers to estimate network reliability with quantifiable confidence while avoiding exhaustive enumeration. The number of samples required to achieve statistically valid coverage is determined using the standard finite population sampling equation, also known as the DATE09 statistical sampling formula:

$$n = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 p(1-p)}}$$

where N is the total number of candidate fault locations, e is the allowable margin of error, t is the value corresponding to the desired confidence level, and p is the estimated probability of a fault occurrence. This formulation accounts for finite population correction, ensuring that the selected sample size provides statistically representative coverage of the entire fault space.[13]

By applying this sampling approach, fault injection experiments can balance accuracy and computational efficiency, ensuring that reliability estimates remain valid even when only a subset of all possible faults is tested [6]. Overall, simulation based and statistically guided fault injection offers a reproducible and scalable methodology for analyzing how SNNs respond to diverse hardware level fault mechanisms.

2.2.3 Fault Injection During Training

Fault injection in SNNs can be performed at different stages of the learning process, each providing complementary insights into the network's reliability. Injecting faults during training examines the system's ability to adapt under faulty conditions,

while after-training fault injection evaluates how well a trained and fixed model maintains performance once learning [14].

During-training fault injection, also referred to as fault-aware training, uses the adaptive learning behavior of SNNs to improve tolerance to hardware imperfections. When faults are introduced concurrently with learning, SNNs can dynamically adjust synaptic weights, firing thresholds, or membrane parameters to compensate for malfunctioning neurons or synapses [14]. This adaptive process exploits mechanisms such as STDP and surrogate-gradient learning, which allow the network to reconfigure itself during training and learn to operate under imperfect hardware conditions [15]. Empirical studies show that networks trained in the presence of controlled perturbations exhibit improved post-deployment fault tolerance, as they implicitly learn to compensate for unreliable components [16] [15].

This adaptive behavior parallels biological neuroplasticity, where damaged synapses or neurons are functionally bypassed through local reorganization of connectivity. Fault-aware training can reduce the effect of hardware variations, slow changes in synaptic weights, and stuck-at defects by retraining around faulty elements [16]. However, the ability to compensate depends on the type and duration of the fault. Temporary faults can often be corrected during learning. In contrast, permanent defects tend to cause lasting performance decrease that cannot be recovered, even when the network adapts during training [15].

By contrast, after training fault injection evaluates system robustness once the learning phase has been completed and model parameters are frozen. In this case, the network no longer adapts, so injected faults directly affect inference accuracy [15]. Experimental results indicate that SNNs trained using STDP or surrogate gradient methods maintain their performance better than conventional ANNs when exposed to post-training faults [16]. Nevertheless, severe or permanent hardware faults can produce persistent misfiring or dead zones that propagate through the network and cannot be mitigated without retraining [15].

During training fault injection measures the system's ability to adapt and recover while after training fault injection measures the ability to maintain correct operation without adaptation. Combining both approaches provides a comprehensive view of the fault tolerance of neuromorphic systems and helps guide the joint development of robust learning algorithms and reliable hardware designs [17].

2.3 Fault Types

Faults in neuromorphic hardware can manifest in multiple ways depending on their physical cause, temporal duration, and functional impact on neuron or synapse behavior. At a high level, hardware faults may alter the electrical state, timing, or functional connectivity of neurons and synapses, leading to abnormal network

activity. Common fault manifestations include dead or inactive neurons, which never fire due to disrupted membrane updates; saturated neurons, which remain constantly active regardless of input; and timing variations, where delays or jitter alter spike timing precision. Other effects such as synaptic or weight corruption change connection strength or polarity, while parametric drifts cause gradual deviations in leak constants or firing thresholds.

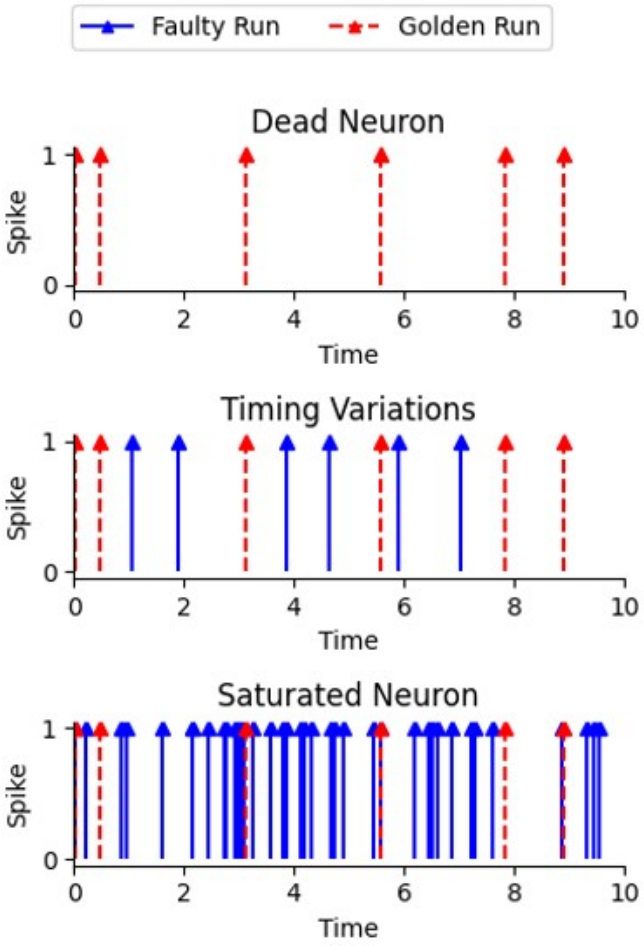

Figure 2.3: Illustration of common fault behaviors in spiking neurons: (top) dead neuron, (middle) timing variations, (bottom) saturated neuron. Red dashed lines indicate fault-free (golden) spikes, and blue solid lines represent faulty behavior [10].

Figure 2.3 illustrates typical examples of these fault behaviors compared to fault-free (golden) neuron activity. In the "dead neuron" case, no spikes are generated despite stimulation, in "timing variations" spike events occur at irregular or shifted

intervals and in "saturated neurons" continuous firing is observed regardless of the input pattern. These behaviors represent distinct manifestations of underlying hardware defects that can propagate through the network and disrupt functionality.

These behavioral anomalies can originate from different physical mechanisms and are generally classified into two main categories: transient faults and permanent faults [18]. The distinction between these two lies primarily in their duration, recurrence, and persistence within the circuit. Transient faults are temporary disturbances that may resolve spontaneously, while permanent faults represent irreversible defects that persist throughout system operation.

2.3.1 Permanent Faults

Permanent faults, also referred to as hard faults, are irreversible hardware defects that persist throughout operation. They typically originate from manufacturing defects, transistor wear-out, electromigration, or prolonged exposure to stress factors such as heat or voltage overstress [18]. Once a component becomes permanently damaged, its electrical behavior deviates from the intended logic, affecting the circuit's long-term functionality. Depending on the affected site, these faults can lead to various behaviors such as dead neurons that never fire, saturated neurons that remain constantly active, or frozen synapses with weights that never update during learning.

Among permanent faults, stuck-at faults represent the most commonly studied model due to their prevalence in digital and neuromorphic circuits. In a stuck-at fault, a signal line, neuron output, or synaptic weight is permanently fixed to a logical '1' or '0' [19]. This type of fault may occur within neuron circuits where a membrane potential fails to update or reset correctly or within synaptic arrays where a weight value remains constant regardless of pre-synaptic activity. Stuck-at faults are particularly detrimental because they persist across all time steps, resulting in permanently inactive or continuously firing neurons that distort information flow through the network. Even a small fraction of such faults can significantly reduce classification accuracy [19]. At higher fault densities, global network instability and biased firing patterns may arise, which cannot be corrected without retraining. Because of their persistent nature, stuck-at faults define the worst-case reliability limit of neuromorphic architectures and are therefore widely used as a benchmark in resilience evaluation studies.

2.3.2 Transient Faults

Transient faults, also known as soft errors, are temporary disturbances that alter circuit behavior for a brief period before the system returns to normal operation. They are primarily caused by external or operational effects such as cosmic radiation,

thermal noise, voltage fluctuations, or timing glitches [18]. In neuromorphic systems, these transient effects can disturb signal transmission, modify stored logic states, or briefly corrupt analog parameters such as membrane potential or synaptic current. Other forms of transient behavior include timing variations, where propagation delays cause spikes to occur earlier or later than expected, and parametric fluctuations, where neuron thresholds or leak constants deviate slightly due to electrical noise.

The most representative model of transient behavior is the bit-flip fault, which inverts a single bit in the binary representation of a stored parameter. For example, a bit-flip in the memory cell of a synaptic weight or neuron threshold changes the encoded value from 0 to 1 or vice versa, corrupting the numerical precision of the parameter [20]. Although such events are short-lived, they can have persistent effects in SNNs due to the network's temporal dynamics—small perturbations may propagate over time, subtly altering firing rates or spike patterns across layers. This cumulative effect can degrade the output distribution even after the transient error disappears [20].

Transient faults are especially relevant for SNNs deployed in radiation-prone or low-voltage environments, such as space applications or edge AI devices, where the probability of soft errors is significantly higher. To assess their impact, researchers often employ SFI, introducing random bit-flip events at controlled rates to model realistic fault occurrences. These studies aim to quantify network sensitivity and determine which parameters, layers, or neuron populations contribute most to error propagation under transient fault conditions [11].

Chapter 3

Methodology

In this chapter, the methodological approach taken to train a resilient SNN model is described. The process involves preparing the datasets, designing the model architecture, developing a fault injecting strategy and establishing the training of SNN.

3.1 Data Preprocessing

NMNIST dataset was denoised using a temporal filter with a window of 10,000 µs to isolate the events. The denoised data was converted into 100 time bins using the sensor size through a frame-based transformation to generate spike-frame representations suitable to train SNN.

To prepare the **AudioMNIST** dataset, raw audio samples were standardized to a fixed duration of 0.73 seconds (35,000 samples at 48 kHz) via truncation or zero padding which is extending shorter signals by appending zeros so that all samples share a consistent length. This guarantees uniform input size across dataset. Each standardized audio sample was then converted into a representation that an SNN can process more easily. Raw audio is simply a one-dimensional waveform showing how air pressure changes over time, but this format does not explicitly show how the energy of the sound is distributed across different frequencies. To extract this information, a Mel filterbank transform was applied. The Mel filterbank first splits the audio into many small overlapping time windows. For each window, it analyzes how much energy the signal contains at different frequencies. These measurements are then grouped into 40 "Mel bands" which are frequency ranges spaced according to how humans perceive differences in frequency [5]. Performing this operation across the entire signal produces a Mel-spectrogram, a two dimensional map in which one axis represents time and the other represents frequency. Each value in the map indicates the strength of the sound energy in a specific band at a given

moment. In this work, windows of 25 ms with a 10 ms step between windows were used, resulting in roughly 71 time frames for each sample. To make the values easier to interpret, the spectrogram was converted from raw amplitude values to the decibel (dB) scale, which compresses the dynamic range and corresponds more closely to perceived loudness. Finally, each spectrogram was normalized independently so that differences in recording volume do not affect the model and all samples occupy a similar numeric range. To have data suitable for SNNs, the normalized Mel-spectrogram values were binarized using a fixed threshold (T=0.9) producing spike based inputs that represent temporal activations across frequency bands. Finally, features were arranged by time steps so that each time frame corresponds to one SNN step.

Each sample in **SHD** consists of spike events distributed across input neurons, where the timing and location of each event correspond to frequency specific responses from a simulated auditory system. To make the data suitable for SNN processing, it was converted the raw spike events into fixed-length frame representations. This transform divides the continuous spike stream into a fixed number of temporal bins which was 100 for this work while preserving the temporal structure of the events. The resulting tensors have dimensions that represent the number of time steps, batch size, and input neurons.

The AudioMNIST dataset were split into training, validation, and test sets (70%, 20%, and 10%). The SHD and NMNIST data were in train and test sets so the one-eighth of the train set randomly split into validation set. Finally, data loaders were defined for each subset to enable efficient batching, shuffling and parallel data processing.

3.2 Model Architecture

For the NMNIST dataset, a model was implemented as a two layer fully connected SNN built with LIF neurons. More biologically detailed neuron models such as adaptive or conductance based LIF could in principle capture richer dynamics however they are computationally more complex than the standard LIF and are less commonly used in large SNN architectures [4]. The standard LIF neuron was adopted as it provides a practical trade-off between computational tractability for implementation in large scale SNNs and sufficient biological realism to allow key intrinsic dynamics, such as membrane decay (β) and firing threshold, to be treated as learnable parameters. The network receives $17 \times 17 \times 2$ event-based frames as input and projects them to a hidden layer of 256 neurons, followed by an output layer of 10 neurons corresponding to the digit classes. Training is performed with MSE count loss and the Adam optimizer (learning rate $=1 \times 10^{-4}$).

A network initialized to be trained by AudioMNIST dataset follows a three layer

fully connected architecture that processes Mel-spectrogram-based spike inputs. Each layer consists of a linear transformation followed by a LIF neuron layer, which models temporal integration and spike-based activation. The architecture comprises an input layer of 40 units. The hidden layers contain 150 neurons, and both the membrane decay constant (β) and firing threshold are trainable, allowing the network to adapt its temporal dynamics during learning. The output layer consists of 10 LIF neurons for digit classification. Training is conducted with MSE count loss and the Adam optimizer (learning rate = 3×10^{-4}). The model trained with SHD is a multi-layer fully connected SNN. It consists of four feedforward layers connected through LIF units, each parameterized by trainable membrane decay constants (β) and firing thresholds (V_{th}). The hidden layers contain 170 neurons and the output layer contains 20 neurons corresponding to the digit classes. Each LIF layer captures temporal dependencies by maintaining membrane potential states across time steps, thereby enabling the model to integrate incoming spikes and generate dynamic spiking outputs. Training uses MSE count loss and the Adam optimizer (learning rate =1×10^{-4}).

Table 3.1: Model Architecture Summary

Model	Input Layer Neurons	Number of Hidden Layers	Neurons per Hidden Layer	Output Layer Neurons
NMNIST	578	1	256	10
AudioMNIST	40	2	150	10
SHD	700	3	170	20

3.3 Fault Injection Strategy

A comprehensive fault injection framework targeting both learnable and run-time parameters is developed. The framework is designed to simulate realistic low-level faults which may occur in digital hardware due to transient errors or permanent defects.

Faults are introduced at the bit level, specifically targeting the 32-bit floating-point representation of the selected value. Two distinct fault models were implemented within the injection framework to represent transient and permanent hardware errors. In the transient fault scenario, a randomly selected bit within a chosen parameter is flipped at the randomly chosen batch time step and epoch. Since these faults are momentary, the affected value can subsequently be updated or corrected during training through backpropagation. In contrast, a stuck-at fault is modeled as a permanent defect, where the selected bit is fixed to logic '0' or

'1' at the start of training and remains unchanged throughout the entire process. This distinction allows the framework to capture both short-lived disturbances and persistent hardware malfunctions within the same experimental setting.

To define the complete search space for fault injection during the training process, a comprehensive candidate list is generated. This list enumerates every potential fault site encountered during a single batch pass, targeting both the static, learnable parameters and the dynamic, run-time variables of SNN. The full set of injection targets, detailed in Table 3.2 and it includes weight elements, gradient entries, per-time-step inputs, linear pre-activations, LIF membrane potentials, membrane time constants (β), and firing thresholds. From this pool, the injector draws a number of targets uniformly at random, ensuring that each parameter element has an equal probability of being selected for corruption. The total length of the candidate list defines the complete search space.

Table 3.2: Comprehensive Candidate List for Fault Injection

Target Type	**Description and Role in the SNN**
Static/Learnable Parameters	
Weight Elements	The fundamental parameters of the network updated during training
Gradient Entries	The calculated error derivatives used during backpropagation to update weights
LIF Firing Thresholds (V_{th})	The voltage threshold that triggers a spike
LIF Membrane Time Constants (β)	The decay rate of the membrane potential
Dynamic/Run-time Variables	
Per-Time-Step Inputs	The input tensor fed into the network at each time step
Linear Pre-activations	The output of the linear transformation before being processed by the LIF neuron
LIF Membrane Potentials	The integrated potential state of the neuron maintained across time steps

By using the Statistical Fault Injection (SFI) sampling formula [13] together with the desired error margin, confidence level, and assumed fault probability, the statistically justified number of fault injections was determined. In this study, the

total population size N corresponds to the number of all possible fault locations in the model, including every element of the weight tensors, gradient tensors, input spikes, activations, membrane potentials, and neuron-specific parameters. The SFI formula ensures that the selected sample size provides unbiased and representative coverage across this population. The resulting value specifies how many faulty samples must be analyzed to obtain statistically meaningful conclusions about resilience. Instead of injecting a single fault into each model, which would require an impractically large number of training sessions, multiple faults were introduced within each run to balance computational efficiency with representativeness. The chosen injection counts for the two distinct fault models reflect the specific nature and typical distribution of the corresponding hardware errors.

In the bit-flip scenario, hundreds of random bit inversions were injected per model. Applying faults in large batches is essential not only for achieving statistically meaningful coverage across the extensive space of runtime and learnable parameters, but also for accelerating the overall resilience analysis [21]. Because transient faults manifest unpredictably and depend strongly on timing, wide and dense sampling of the parameter space is required to capture their full range of effects. Injecting many faults within each epoch therefore provides a practical and efficient means of evaluating the network's sensitivity to non-deterministic, time-dependent failures[21].

Conversely, in the stuck-at case, only three randomly selected bits were forced to fixed 0. Stuck-at faults model permanent defects, such as those caused by manufacturing flaws or device aging. Such defects are generally sparse yet have a catastrophic, permanent impact where they occur [21]. This low fault count is intentionally selected to simulate the failure of a limited number of critical elements. Three faults are injected instead of a single fault to ensure the experiment evaluates the network's generalized defect tolerance across different sparse locations, thereby providing results with increased statistical robustness compared to isolating the effect of a single failure event. Consequently, the network's capacity for defect tolerance against multiple stuck-at faults is assessed while maintaining computational feasibility and accurately representing the physical reality of hardware defects.

For each experiment, candidate fault locations are systematically identified. A subset is then randomly selected for injection, ensuring proper coverage and randomness. During model training, at each batch and time step, the framework checks if any pending faults are scheduled for injection. When a fault condition is met, the corresponding value is perturbed using the selected bit-level operation. As shown in Figure 3.4, the framework supports fault injection targeting two main parameter categories. The learnable parameter (weights, gradients and neuron parameters β and V_{th}) faults are injected after backward pass. The run time variable (inputs, activations, and membrane potentials) faults are injected during forward pass at right time step batch and epoch. The original and corrupted values,

Algorithm 1 Training-Time Fault Injection Procedure

```
Input: SNN model M, datasets
Output: Golden accuracy, faulty accuracies
Train M without faults to obtain the golden model accuracy.
Generate fault candidate list (weights, gradients, β, threshold, input, activation, membrane).
Compute number of faulty models via DATE09 formula (95% confidence, 5% error).
for each sampled faulty model do
    if stuck-at experiment then
        Select 3 random fault locations.
        for each epoch do
            Forward pass with input/activation/membrane faults.
            Backpropagation.
            Inject stuck-at faults with weights/gradients/β/thresholds.
        end for
        Evaluate inference accuracy vs. golden model.
    else
        Select ∼300 bit-flip faults, distribute across epochs.
        for each epoch do
            Forward pass with input/activation/membrane faults.
            Backpropagation.
            Inject stuck-at faults with weights/gradients/β/thresholds.
            Record validation accuracy vs. golden model.
        end for
    end if
end for
```

along with metadata such as the bit index, layer, and parameter location, are logged. By logging all injected faults and their effects, the framework enables a thorough evaluation of the SNN's fault tolerance, revealing both immediate and accumulating impacts.

This methodology allows for a controlled, repeatable, and statistically justified exploration of SNN vulnerability to hardware-level errors, and provides insights into the resilience of spiking neural network models under realistic deployment conditions.

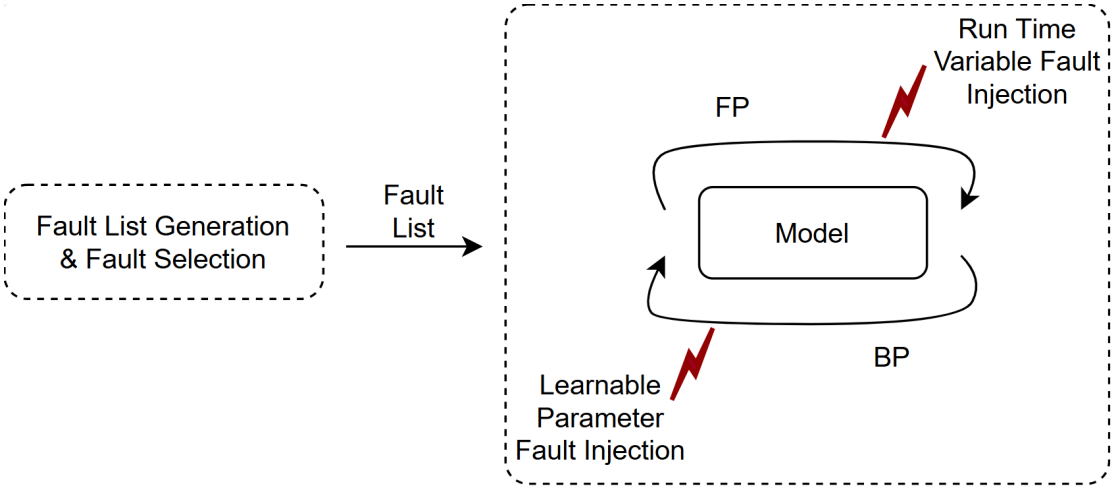

Figure 3.1: Fault Injection Diagram

3.4 Training Procedure

As shown in Figure 3.5, the network is first trained under fault-free conditions to establish a baseline performance reference before introducing any faults. Following this stage, a series of fault injected training runs are conducted. The number of models is determined using the previously described statistical sampling method. To ensure that each faulty model represents a consistent perturbation of the golden model, a fixed random seed is used to control weight initialization, data shuffling, and other stochastic components of training. This guarantees that the only difference between the golden model and each faulty model is the injected faults themselves. For each run, a distinct set of randomly selected fault locations is applied, providing diverse coverage of potential fault scenarios across models. Throughout all runs, the framework enforces the correct fault-injection schedule, and every injected fault together with the corresponding model outputs is recorded in detailed logs for subsequent analysis and comparison against the baseline model.

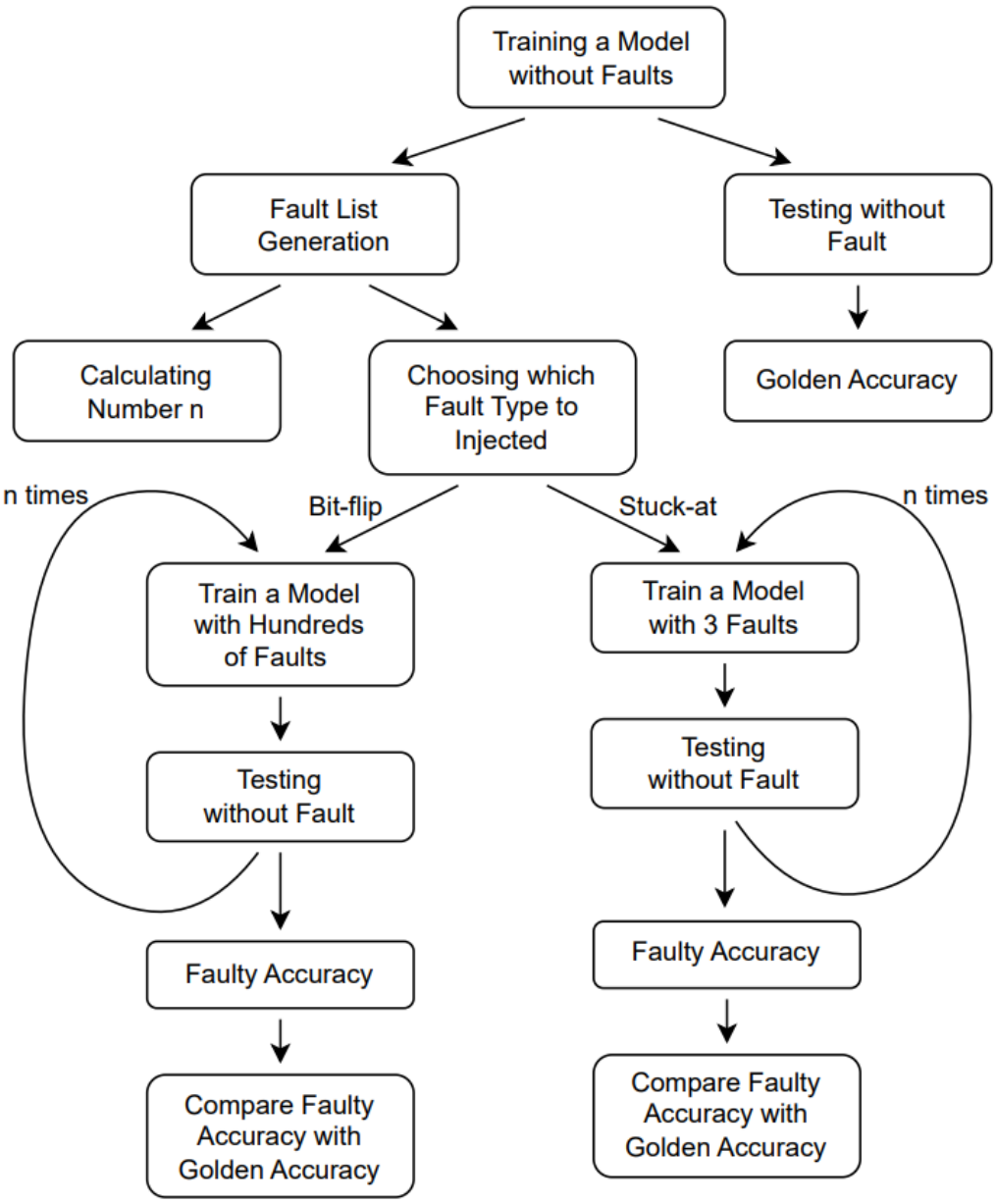

Figure 3.2: Training Diagram

Chapter 4

Experiments and Results

4.1 Datasets

This section introduces three fundamental benchmark datasets NMNIST, AudioMNIST, and Spiking Heidelberg Digits (SHD). The following subsections detail the unique generation methodologies, structural characteristics, and research utility of each dataset.

4.1.1 NMNIST

The NMNIST dataset is a neuromorphic, event based version of the classical MNIST dataset for handwritten digits. It was developed to provide a standardized and biologically inspired benchmark for SNN research, addressing the lack of large, publicly available datasets compatible with neuromorphic vision sensors [22]. It retains the familiarity and simplicity of MNIST while enabling direct evaluation of spike based algorithms under realistic, asynchronous sensory input conditions.

To generate NMNIST, an ATIS and a DVS type camera that records changes in pixel intensity rather than absolute brightness values is used [22]. Each pixel in the ATIS operates independently, producing a stream of discrete events whenever a change in luminance occurs. An event encodes four pieces of information which are the pixel's x-coordinate, y-coordinate, the polarity of the brightness change, and a timestamp with microsecond precision. Instead of moving the images on a monitor, the authors physically moved the sensor itself using an automated pan–tilt platform [22]. This movement reproduced three rapid, small amplitude eye-like motions tracing a triangular path, each lasting approximately 100 milliseconds. During each motion, the camera recorded the asynchronous stream of ON and OFF events triggered by changes in pixel intensity caused by the relative motion between the static image and the moving sensor.

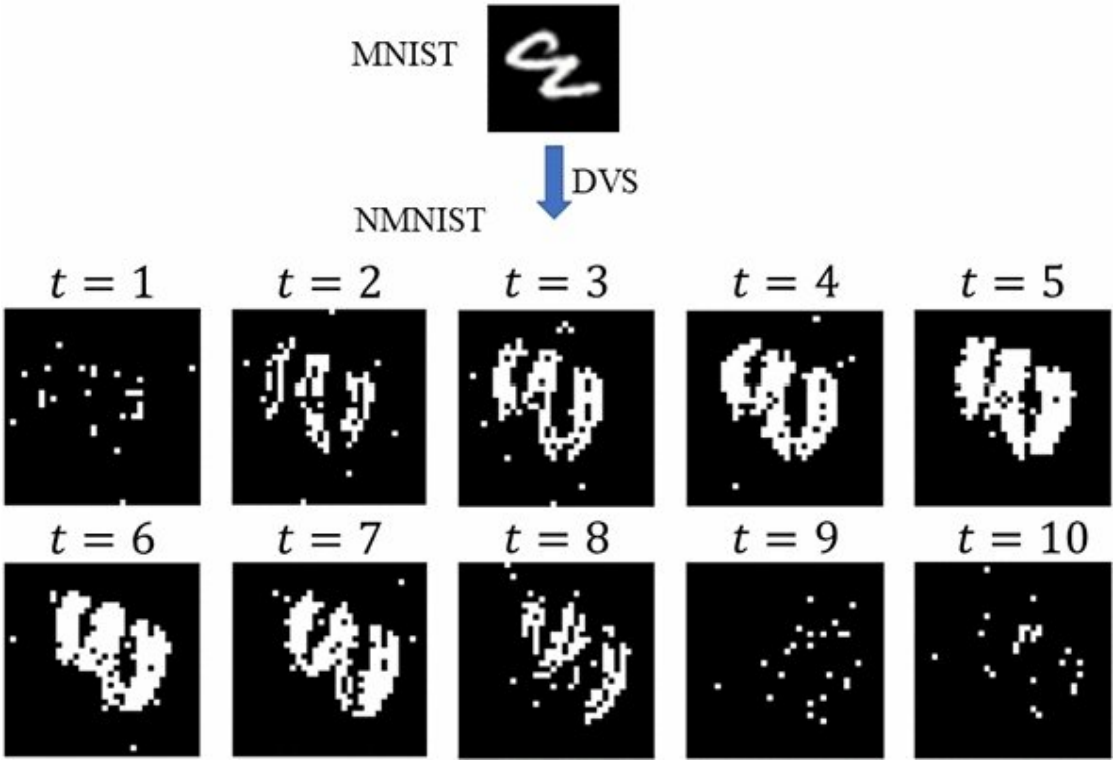

Figure 4.1: NMNIST Dataset [23]

This design choice avoided the temporal artifacts that arise when simulating motion on computer screens and ensured that the recordings reflected real, continuous motion with biologically realistic noise characteristics. Each digit from the original MNIST dataset was displayed on an LCD screen and recorded under identical lighting and distance conditions. The complete dataset of 70,000 images was converted. Each NMNIST sample corresponds to one handwritten digit from the original MNIST dataset, but instead of a static 28×28 grayscale image, it is represented as a spatio-temporal spike event stream lasting about 300 milliseconds [22]. The spatial resolution remains 28×28 pixels to maintain consistency with the original dataset, ensuring easy comparison across conventional and neuromorphic models.

The resulting dataset retains the 10 class structure of MNIST but introduces an additional temporal dimension and polarity channel [22]. On average, each recording contains roughly 4,000 events, with approximately equal proportions of ON and OFF events. This temporal structure makes NMNIST well-suited for algorithms that exploit timing, such as spiking neurons.

4.1.2 AudioMNIST

The AudioMNIST dataset is a publicly available speech dataset designed for controlled experimentation in audio classification [24]. It was introduced as a simple and carefully structured benchmark intended to play a role in the audio domain similar to that of the MNIST dataset in computer vision. While MNIST provides a standardized platform for evaluating image classifiers, AudioMNIST enables systematic investigation of machine learning models on spoken digit recognition and related acoustic tasks under well-defined conditions. The primary motivation behind creating AudioMNIST was to establish a clean, small-scale, and well-balanced dataset for studying how deep neural networks process, represent, and interpret audio signals [24]. Existing large-scale speech datasets are often too complex for controlled analysis, interpretability studies, or robustness testing. AudioMNIST focuses on a narrow and clearly defined task that is recognition of digits zero through nine allowing researchers to analyze model behavior without background noise, or uncontrolled recording environments. Furthermore, AudioMNIST is designed to support the development and evaluation of explainable artificial intelligence methods for the audio domain.

AudioMNIST consists of 30,000 recordings, corresponding to approximately 9.5 hours of speech data. The dataset includes audio samples of digits 0 to 9, spoken 50 times each by 60 speakers resulting in a balanced and uniform dataset across all ten digit classes [24]. Recordings were conducted in quiet office environments to minimize background noise and ensure consistent audio quality. Speech samples were captured using a RØDE NT-USB microphone as mono-channel signals with a sampling frequency of 48 kHz, and stored in 16-bit integer format [24]. The resulting files are single word audio sample of digits pronounced in English, offering clear acoustic boundaries and minimal temporal overlap.

AudioMNIST supports a broad spectrum of research directions, ranging from speech recognition and robustness analysis, and neuromorphic computation [24]. Its simplicity, balanced structure, and high quality recordings make it particularly suitable for studies requiring controlled experimental conditions, including evaluating model robustness against noise, quantization errors, or hardware induced faults.

4.1.3 SHD

The SHD dataset is one of the benchmark datasets that was specifically designed for the systematic evaluation of SNNs, providing a biologically realistic and standardized dataset for auditory pattern recognition tasks. Unlike event converted datasets such as NMNIST, SHD consists of spike sequences generated from real speech recordings, making it a natural and temporally precise benchmark for testing the capabilities of time based neural computation. It is modeled that how a human

ear would respond to natural auditory input, producing spikes directly as the output of a simulated human ear. This approach enables the dataset to combine biological plausibility with machine learning practicality, allowing fair comparison between conventional artificial neural networks and spiking models on equivalent classification tasks [25].

The SHD dataset is derived from recordings of spoken digits from zero to nine, produced by 12 male and 12 female speakers. Each audio sample was preprocessed through a simulated human ear model, resulting in a temporally precise stream of spikes distributed over 700 frequency channels [25]. These channels correspond to the outputs of different frequency bands in the human ear filter bank, which encodes how the human auditory system decomposes complex sounds across both time and frequency. The dataset contains 8156 training samples and 2264 test samples, each lasting approximately 1 second [25]. Every sample is represented as a set of spike times and their corresponding channel indices. In contrast to datasets that require pre-processing steps such as feature extraction or time window slicing, the SHD dataset provides spike level data, removing the need for artificial conversion between continuous and discrete representations [25].

A key feature of SHD is its biophysical realism and this makes it valuable for investigating temporal coding and spike time learning rules, fault injection and resilience under hardware inspired errors, spike based sequence classification, and comparisons between SNN and ANN based recurrent or convolutional models using identical input content. Additionally, it serves as an essential tool for testing how neural architectures handle noise, delay, and fault perturbations in temporal spike data.

4.2 Experiments

For each dataset, experiments were conducted under two hardware fault models: permanent stuck-at faults and transient bit-flip faults. A fault-free baseline model was first trained for each dataset to establish a reference for comparison. After training, the total number of fault susceptible elements was quantified. Since faults are injected at the bit level rather than per numerical parameter, the total fault space is defined as the number of parameter elements multiplied by the bit width of their floating-point representation. This count is further multiplied by the number of simulation time steps. Based on the computed fault space, the number of required fault-injected models was determined using statistical sampling with 95% confidence level and a 5% margin of error, ensuring representative coverage without exhaustive enumeration. Each sampled model instance was then trained independently under injected faults to analyze fault propagation across distinct fault locations. Based on the computed fault space and statistical sampling criteria,

Table 4.1: Parameter counts and sampled fault space for each dataset.

Parameter Type	**NMNIST**	**AudioMNIST**	**SHD**
Weights	150528	30000	180200
Gradients	151060	30620	181260
β	266	310	530
Threshold	266	310	530
Input	106752	43520	38720
Activation	34048	39680	16960
Membrane	34048	39680	16960
Total	1526297600	589184000	1392512000
Sample Size	384	384	384

a total of 384 fault injected training runs were carried out for each dataset. For the stuck-at fault experiments, three fault locations were randomly selected prior to the start of each training run. The selected bits were permanently forced to zero from the first epoch until the end of training, emulating irreversible hardware defects such as persistent memory corruption. After training, the resulting inference accuracy for each faulty model was recorded and compared against the corresponding golden model to quantify degradation in final performance.

For the bit-flip experiments, each model was trained under approximately 300 transient faults. Instead of injecting all faults at once, faults were evenly distributed across epochs such that the number of injected faults per epoch equaled the total fault count divided by the number of training epochs. Each fault was assigned a specific epoch, batch, and time step, ensuring precise injection timing during training. Validation accuracy was recorded after each epoch, and the accuracies corresponding to epochs in which faults were injected were compared against the fault free validation trajectory of the golden model to analyze the impact of transient faults on learning dynamics.

4.3 Results

Since faults were injected by selecting parameters uniformly based on the injectable space at random, the distribution of observed faults naturally reflected the parameter counts of each component. As expected, input, activation, and membrane variables—being the most numerous—appeared most frequently, whereas beta and threshold parameters were encountered far less often due to their small parameter

footprints. Weight and gradient faults were expected to occur at intermediate frequencies; however, gradient-related faults appeared too rarely to support meaningful conclusions.

To quantify the effect of the different fault characteristics on model robustness, the accuracy recorded after each injected fault was compared with the corresponding fault-free accuracy, and the resulting differences were aggregated for each fault configuration. Importantly, these differences were not always negative. In several cases, fault injections produced a slight increase in accuracy. This behavior is consistent with known learning dynamics. Certain perturbations can unintentionally act as a form of stochastic regularization, disrupting overfitted parameter configurations and thereby improving generalization on the validation set. In other words, some faults introduce noise that counteracts overfitting, leading to higher post-fault accuracy.

These aggregated deviations were then visualized across multiple dimensions, including fault type, bit position, layer location, and injection epoch, allowing us to characterize how different fault attributes influence overall model sensitivity.

4.3.1 Bit-Flip Fault

The experiments conducted on the AudioMNIST dataset reveal that the accuracy degradation caused by bit-flip faults is strongly dependent on the epoch at which the fault is injected. Faults introduced during the early stages of training tend to produce only minor changes in accuracy, as the model parameters are still highly dynamic and have not yet converged. In contrast, faults injected in later epochs lead to substantially larger drops in accuracy, indicating that the network becomes increasingly sensitive to faults as it approaches convergence. This behavior is consistent with the underlying optimization dynamics: early in training, large gradient magnitudes cause substantial parameter updates that can wash out the effect of small disruptions, whereas in later epochs the gradients become smaller and the loss landscape stabilizes, so injected faults are no longer overridden by subsequent learning steps. Moreover, in some cases the injected fault altered the optimization trajectory so severely that the model failed to continue meaningful learning altogether, leading to abrupt plateaus in validation accuracy. As illustrated in Figure 4.2., all fault types exhibit a similar trend: the negative impact on accuracy consistently grows with the epoch index. This suggests that once the model has formed a stable representation, even small numerical changes can significantly impair performance, highlighting the importance of considering the temporal dimension of training when assessing the fault resilience of SNNs.

In addition to the temporal sensitivity observed across epochs, the experiments also reveal that the impact of bit-flip faults varies substantially across different fault types. This indicates that the vulnerability of the network is not uniform

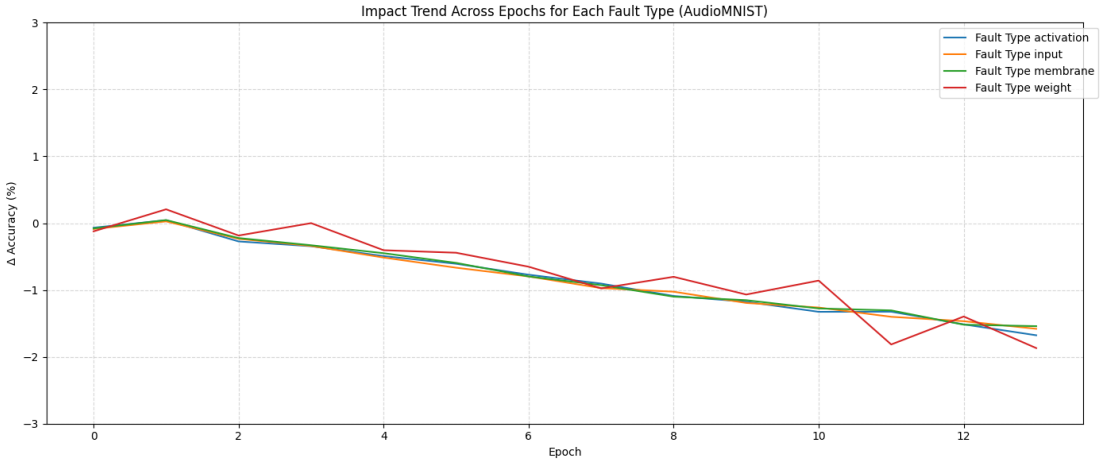

Figure 4.2: Accuracy degradation caused by single bit-flip faults across different epochs and fault types in the AudioMNIST SNN model.

across its internal components, but instead depends on the specific parameter or state variable in which the fault occurs. Figure 4.3 illustrates this effect for the AudioMNIST model, showing the average accuracy degradation produced by single bit-flips across all bit positions and fault types. As the graph demonstrates, bit-flips injected into weight parameters consistently produce larger accuracy drops compared to those injected into activation, membrane, or input variables. This pattern reflects the persistent and global influence of weights throughout inference, whereas faults in transient state variables tend to remain localized and therefore make a weaker overall effect. The relatively smaller impact observed for beta and threshold parameters is primarily explained by their lower parameter count, which leads to fewer fault injection events within these structures. Overall, these results highlight that fault resilience in SNNs is strongly fault type dependent, with weight storage structures emerging as the most influential contributors to accuracy degradation under bit-flip faults.

A similar analysis was conducted on the SHD dataset to further examine how different fault types influence the model's resilience under bit-flip faults. Unlike AudioMNIST, where weight related faults dominated the degradation pattern, the SHD model exposes a more heterogeneous fault type sensitivity profile. As shown in Figure 4.4, beta and threshold faults produce noticeably larger spikes in accuracy degradation for several specific bit positions. This effect becomes more visible in the SHD model because beta and threshold parameters exist in much smaller quantities compared to other parameter types. With fewer parameters available to absorb the disturbance, a single bit-flip in these components modifies a larger fraction of their representational capacity, making the resulting deviation

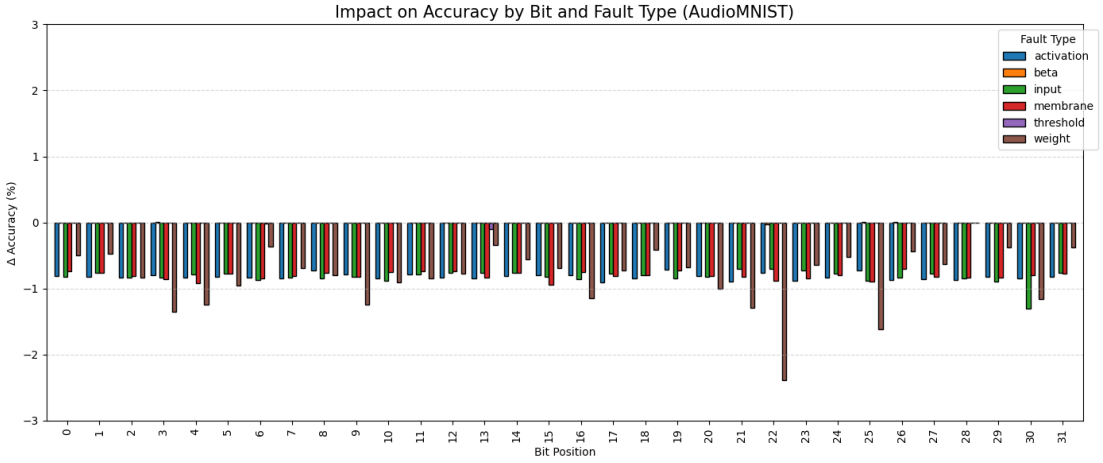

Figure 4.3: Accuracy degradation caused by single bit-flip faults across different bit positions and fault types in the AudioMNIST SNN model.

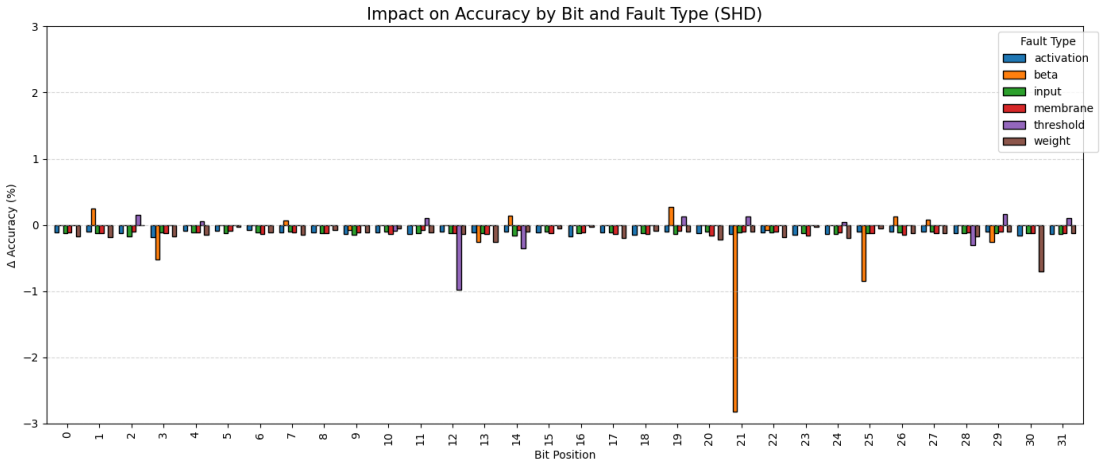

Figure 4.4: Accuracy degradation caused by single bit-flip faults across different bit positions and fault types in the SHD SNN model.

more effective. In contrast, faults injected into weights, activations, inputs, and membrane variables continue to exhibit relatively small and consistent impacts across most bit positions, reflecting the broader distribution of parameters within these categories. Overall, these results show that the most effective fault types are weight, beta, and threshold, which consistently produce the largest accuracy drops across bit positions. These components are more susceptible than activation, membrane, or input variables, making them the primary contributors to accuracy degradation caused by bit-flip faults.

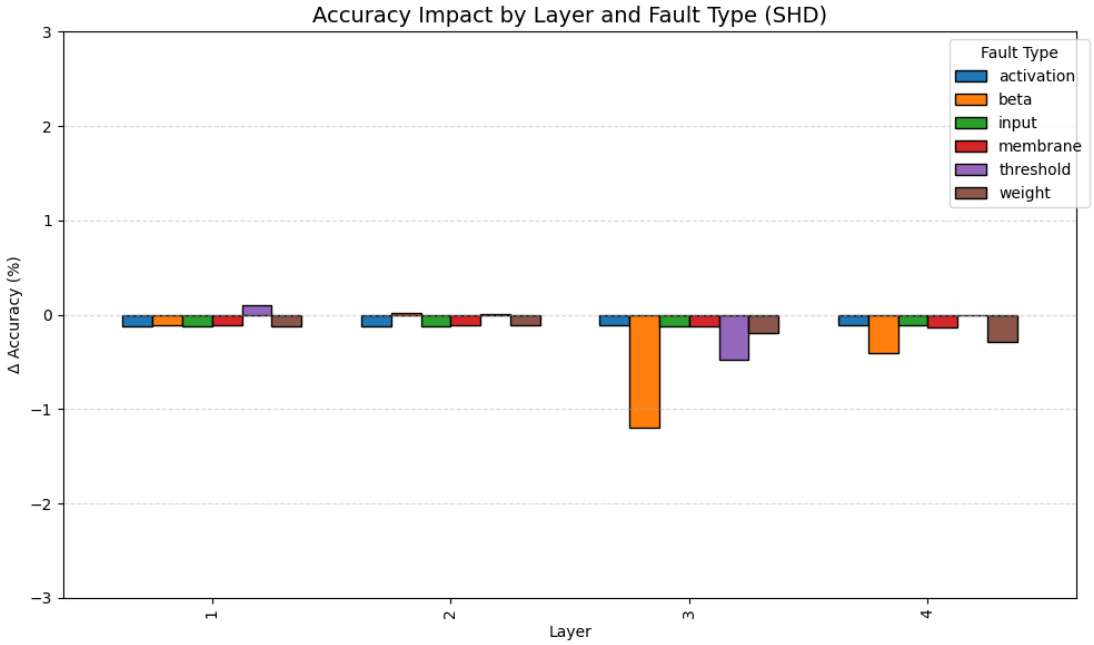

Figure 4.5: Accuracy degradation caused by single bit-flip faults across different layers and fault types in the SHD SNN model.

Moreover, how the sensitivity to bit-flip faults varies across different layers of the network is examined. Figure 4.5 shows that faults injected into the deeper layers (Layer 3 and Layer 4) consistently lead to larger drops in accuracy compared to faults occurring in the earlier layers. This pattern aligns with the functional role of later layers, which typically encode higher level and more discriminative features. As a consequence, faults introduced at these stages directly disrupt the final representations used for classification. In contrast, faults in the earlier layers have a more limited impact, as subsequent layers can partially absorb or compensate for small deviations in intermediate activations. Although the magnitude of degradation still depends on the specific fault type, the overall trend indicates that faults occurring deeper in the SHD model are inherently more harmful, further emphasizing the structural factors that shape fault sensitivity in SNNs.

In contrast to the pronounced sensitivity observed in SHD and AudioMNIST, the NMNIST model exhibits a remarkably stable behavior under the same fault injection conditions. As illustrated in Figure 4.6, faults applied to any layer or parameter type in NMNIST result in accuracy changes that are effectively negligible, remaining close to zero across all configurations. This indicates that the network's intermediate representations and synaptic parameters are highly resilient, such

that injected faults fail to produce meaningful deviations in the model's decision process. While this confirms that fault impact can vary substantially across datasets and architectural choices, the NMNIST results further demonstrate that these differences can be strong enough to completely suppress the fault-related trends observed in other models. In particular, although patterns such as the increasing influence of faults in later epochs and the heightened sensitivity of weight, beta, and threshold parameters appear consistently across several experiments, some architectures remain sufficiently robust that these effects are not visibly expressed, underscoring that fault sensitivity is ultimately model dependent.Overall, our experiments reveal that single bit-flip faults can alter model accuracy by as much as ±3%, depending on the dataset, architectural configuration, and the location of the fault. This shows that although many SNN architectures exhibit strong resilience, bit-level disruptions are still capable of producing non-negligible deviations in performance.

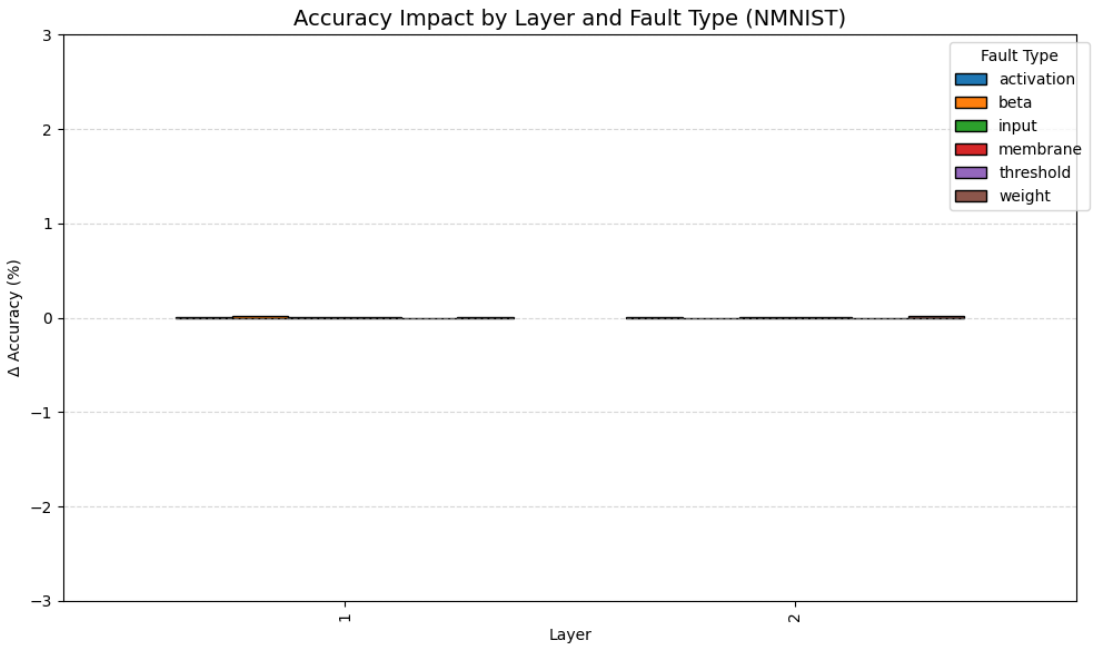

Figure 4.6: Accuracy degradation caused by single bit-flip faults across different layers and fault types in the NMNIST SNN model.

4.3.2 Stuck-At Fault

To further understand how stuck-at faults influence the robustness of SNNs, the average accuracy degradation across layers and bit positions for all three datasets

are analysed. Figures 4.7–4.8 illustrate that, in both AudioMNIST and NMNIST, faults injected into earlier layers have noticeably weaker impact compared to those occurring in deeper layers. Although this trend is less pronounced than in the bit-flip experiments, a consistent pattern emerges. The final layers contribute more strongly to performance degradation, reflecting their role in producing higher level representations that are directly tied to the classification decision.

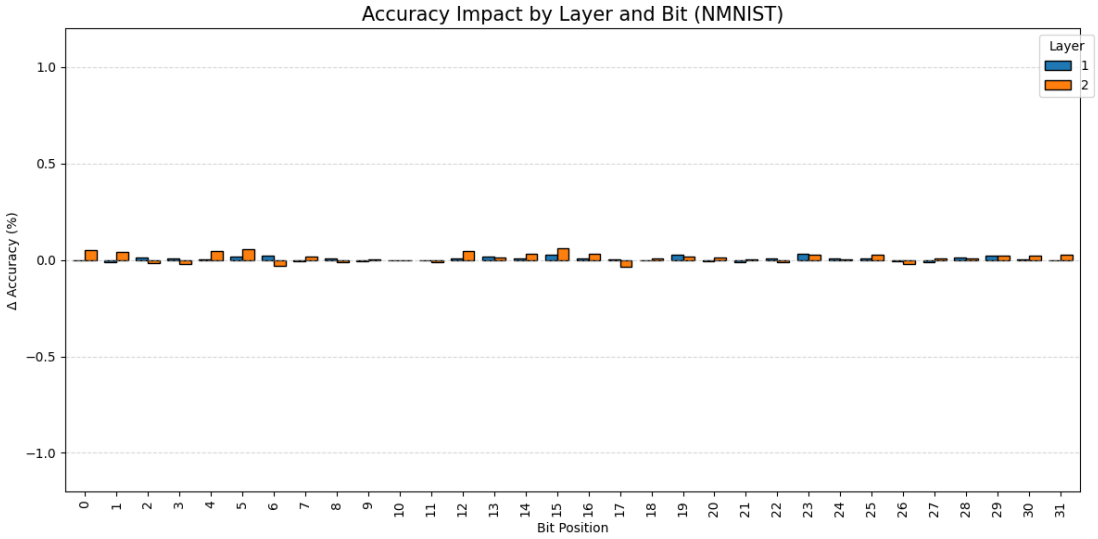

Figure 4.7: Accuracy degradation caused by single stuck-at faults across different layers and bit positions in the NMNIST SNN model.

In the SHD dataset, the behavior differs slightly. As shown in Figure 4.9, the impact does not increase monotonically with the bit index, yet a subtle upward tendency can be observed. Despite local fluctuations, higher index bits tend to induce slightly larger deviations compared to lower index ones. This indicates that stuck-at faults in certain numerical regions of the parameter representation can have disproportionately greater influence, even if the effect is not strictly linear.

Across all datasets, the magnitude of accuracy degradation caused by stuck-at faults is generally modest. It typically remaining within $\pm 1.5\%$. This level of disruption is noticeably smaller than the maximum effects observed under bit-flip faults, yet still sufficient to reveal structural differences between architectures. Taken together, the results show that while some models such as NMNIST remain comparatively resilient, others exhibit clear layer and bit-dependent sensitivity. These findings reinforce that stuck-at faults, despite their lower severity, can expose underlying patterns of vulnerability shaped by model depth, representational hierarchy, and dataset characteristics.

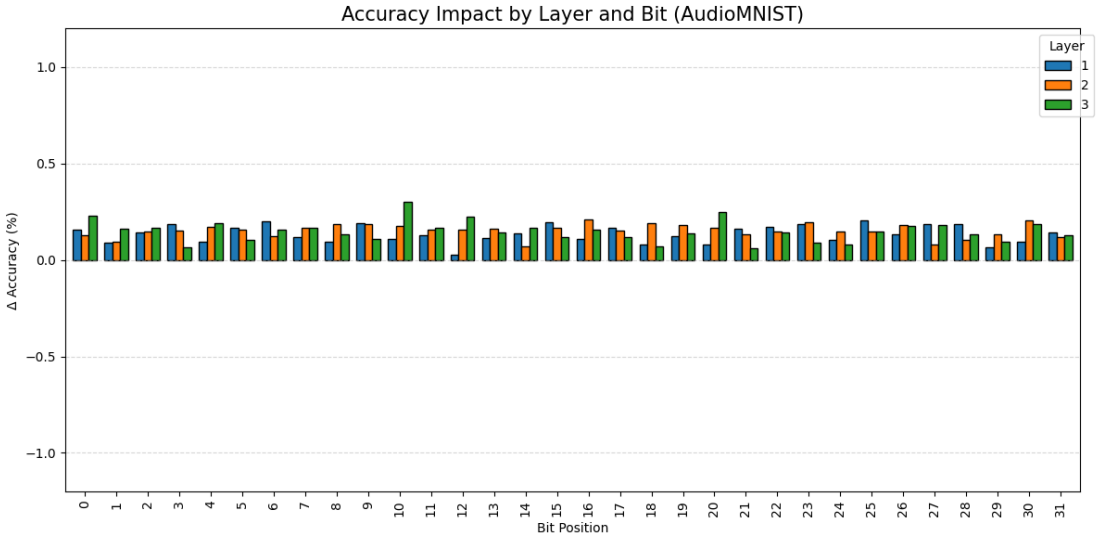

Figure 4.8: Accuracy degradation caused by single stuck-at faults across different bit positions and layers in the AudioMNIST SNN model.

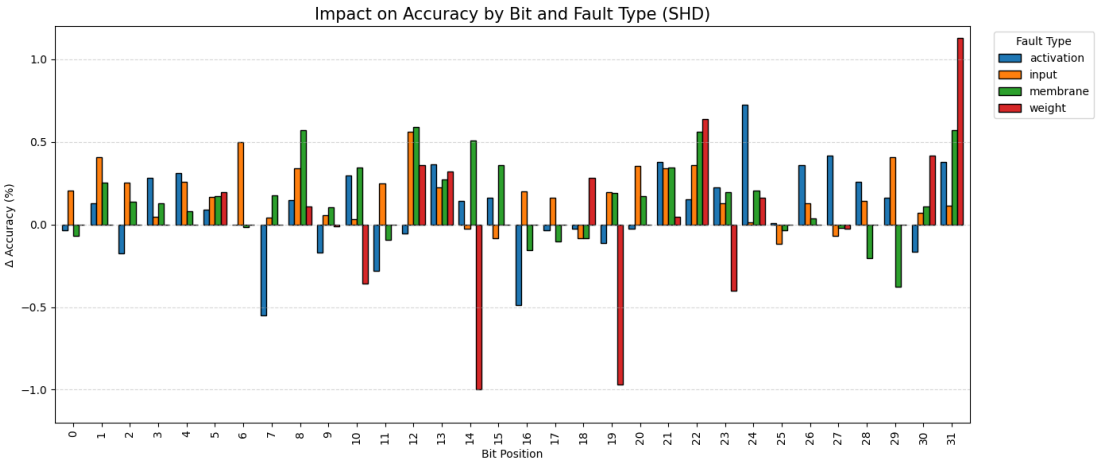

Figure 4.9: Accuracy degradation caused by single stuck-at faults across different bit positions and fault types in the SHD SNN model.

Chapter 5

Conclusion and Future Work

This thesis explored the resilience of SNNs to hardware inspired faults injected during training, addressing the lack of studies examining how learning dynamics interact with internal errors. By injecting faults into weights, gradients, activations, membrane potentials, input events, and neuron-specific parameters such as β and threshold, this work provides a detailed characterization of how different components of an SNN respond to faults that may arise in realistic neuromorphic hardware.

The experimental results reveal several consistent patterns. Faults introduced in the later stages of training produced significantly larger accuracy degradations compared to those applied early on, indicating that networks become increasingly sensitive as their representations stabilize. Certain parameter groups such as weights, β, and threshold emerged as substantially more influential than others. Faults affecting these components produced the most pronounced accuracy drops, while activation, input, and membrane faults tended to have smaller effects due to their transient nature. Layer depth also played an important role. In multiple models, faults applied to later layers resulted in larger deviations, reflecting the greater semantic importance of deeper representations.

Despite these common trends, the experiments also highlighted that fault sensitivity is highly model dependent. While AudioMNIST and SHD exhibited clear vulnerability patterns, the NMNIST model remained remarkably robust, with most injected faults causing negligible degradation. This contrast demonstrates that the resilience of an SNN is shaped not only by the fault type itself but by the specific combination of network architecture, parameter distribution, and dataset characteristics.

Beyond characterizing the fault sensitivity of the evaluated models, the fault injection framework developed in this thesis provides a practical basis for future research on resilient SNN design. Because the injector operates during training and can target a wide range of model components, it enables controlled studies

of how learning dynamics behave under faulty conditions. In doing so, the tool serves as a foundation for systematically developing, evaluating, and benchmarking robust neuromorphic learning systems.

In addition, extending the experimental analysis to a broader range of datasets would help determine the generality of the observed fault resilience patterns. While this thesis focused on three representative neuromorphic benchmarks spanning visual, auditory, and temporal modalities, incorporating datasets with higher spatial complexity, richer temporal structure, or larger class sets would provide a more comprehensive understanding of how data characteristics shape fault behavior.

A second important direction is the exploration of alternative network architectures. The present work evaluated fully connected spiking models, yet modern neuromorphic applications frequently rely on SCNNs, recurrent architectures, and hybrid designs such as SCNNs or spiking attention mechanisms. Investigating these architectures would reveal whether convolutional feature hierarchies, recurrent temporal dependencies, or structured connectivity introduce different sensitivity patterns compared to the FC models considered here. Similarly, expanding the analysis to diverse neuron models such as adaptive LIF neurons, Izhikevich-type dynamics, or multi-compartment neurons may uncover additional fault–response behaviors arising from richer internal dynamics.

Beyond SNNs, the proposed fault injection framework can also be adapted to non-spiking neural models, particularly CNNs. Comparing SNNs and CNNs under identical fault settings would clarify whether spiking dynamics provide inherent resilience advantages or whether the two paradigms exhibit similar vulnerabilities across architectural components. Such cross-paradigm comparisons would be valuable for understanding the robustness trade-offs between spiking and non-spiking computation in fault-prone hardware environments.

Finally, leveraging the injector as part of the training process opens further opportunities. By introducing controlled disturbances into the learning pipeline, the tool enables the exploration of fault-aware optimization strategies based on fault scheduling, and promotes robustness by intentionally exposing the model to disruptive conditions in order to cultivate resilient internal representations. In addition, integrating the injector into hardware-in-the-loop pipelines would allow validation of the simulation based findings directly on neuromorphic processors, ultimately contributing to the development of fault tolerant learning systems suitable for safety-critical or resource constrained deployments.

Bibliography

[1] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. «ReSpawn: Energy-Efficient Fault-Tolerance for Spiking Neural Networks considering Unreliable Memories». In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643524 (cit. on pp. 2, 9).

[2] Karthikeyan Nagarajan, Junde Li, Sina Sayyah Ensan, Mohammad Nasim Imtiaz Khan, Sachhidh Kannan, and Swaroop Ghosh. «Analysis of Power-Oriented Fault Injection Attacks on Spiking Neural Networks». In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2022, pp. 861–866. DOI: 10.23919/DATE54114.2022.9774577 (cit. on pp. 2, 9).

[3] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. «Spiking Neural Networks and Their Applications: A Review». In: *Brain Sciences* 12.7 (2022). ISSN: 2076-3425. DOI: 10.3390/brainsci12070863. URL: https://www.mdpi.com/2076-3425/12/7/863 (cit. on pp. 3–8).

[4] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks». In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022, pp. 14–19. DOI: 10.1109/ISVLSI54635.2022.00016 (cit. on pp. 3, 7, 16).

[5] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker+: A Framework for the Generation of Efficient Spiking Neural Networks FPGA Accelerators for Inference at the Edge». In: *IEEE Transactions on Emerging Topics in Computing* 13.3 (2025), pp. 784–798. DOI: 10.1109/TETC.2024.3511676 (cit. on pp. 3–7, 15).

[6] Anıl Bayram Göğebakan, Enrico Magliano, Alessio Carpegna, Annachiara Ruospo, Alessandro Savino, and Stefano Di Carlo. «SpikingJET: Enhancing Fault Injection for Fully and Convolutional Spiking Neural Networks». In: (2024), pp. 1–7. DOI: 10.1109/IOLTS60994.2024.10616060 (cit. on pp. 4, 5, 7–10).

[7] Rachmad V. W. Putra and Muhammad Shafique. «FSpiNN: An Optimization Framework for Memory- and Energy-Efficient Spiking Neural Networks». In: July 2020. DOI: 10.48550/arXiv.2007.08860 (cit. on p. 5).

[8] Inc. The MathWorks. «Convert Convolutional Network to Spiking Neural Network». In: *MATLAB Help Center – Deep Learning Toolbox Documentation*. Accessed: 24 Nov 2025. 2025 (cit. on p. 6).

[9] Haralampos-G. Stratigopoulos, Theofilos Spyrou, and Spyridon Raptis. «Testing and Reliability of Spiking Neural Networks: A Review of the State-of-the-Art». In: *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2023, pp. 1–8. DOI: 10.1109/DFT59622.2023.10313541 (cit. on pp. 9, 10).

[10] Sarah A. El-Sayed, Theofilos Spyrou, Antonios Pavlidis, Engin Afacan, Luis A. Camuñas-Mesa, Bernabé Linares-Barranco, and Haralampos-G. Stratigopoulos. «Spiking Neuron Hardware-Level Fault Modeling». In: *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2020. DOI: 10.1109/IOLTS50870.2020.9159745 (cit. on pp. 9, 12).

[11] Yi-Zhan Hsieh, Hsiao-Yin Tseng, I-Wei Chiu, and James Chien Mo Li. «Fault Modeling and Testing of Spiking Neural Network Chips». In: *2021 IEEE International Test Conference in Asia (ITC-Asia)*. 2021, pp. 1–6. DOI: 10.1109/ITC-Asia53059.2021.9808431 (cit. on pp. 9, 14).

[12] Jingying Li, Bo Sun, and Xiaoyan Xie. «A Reliability Assessment Approach for A LIF Neurons Based Spiking Neural Network Circuit». In: *2023 24th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE)*. 2023, pp. 1–7. DOI: 10.1109/EuroSimE56861.2023.10100785 (cit. on p. 10).

[13] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. «Statistical fault injection: Quantified error and confidence». In: *2009 Design, Automation Test in Europe Conference Exhibition*. 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716 (cit. on pp. 10, 18).

[14] Catherine D. Schuman, J. Parker Mitchell, J. Travis Johnston, Maryam Parsa, Bill Kay, Prasanna Date, and Robert M. Patton. «Resilience and Robustness of Spiking Neural Networks for Neuromorphic Systems». In: *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020, pp. 1–10. DOI: 10.1109/IJCNN48605.2020.9207560 (cit. on p. 11).

[15] Zalfa Jouni and Haralampos-G. Stratigopoulos. «STDP-Trained Spiking Neural Network Reliability Assessment Through Fault Injections». In: *31st IEEE International Symposium on On-Line Testing and Robust System Design*. Ischia, Italy, July 2025. URL: https://hal.science/hal-05083335 (cit. on p. 11).

[16] Elena-Ioana Vatajelu, Giorgio Di Natale, and Lorena Anghel. «Special Session: Reliability of Hardware-Implemented Spiking Neural Networks (SNN)». In: *2019 IEEE 37th VLSI Test Symposium (VTS)*. 2019, pp. 1–8. DOI: 10.1109/VTS.2019.8758653 (cit. on p. 11).

[17] Atif Hashmi, Hugues Berry, Olivier Temam, and Mikko Lipasti. «Automatic abstraction and fault tolerance in cortical microachitectures». In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 1–10. DOI: 10.1145/2000064.2000066 (cit. on p. 11).

[18] Murat Isik, Ankita Paul, M. Lakshmi Varshika, and Anup Das. «A design methodology for fault-tolerant computing using astrocyte neural networks». In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF '22. Turin, Italy: Association for Computing Machinery, 2022, pp. 169–172. ISBN: 9781450393386. DOI: 10.1145/3528416.3530232. URL: https://doi.org/10.1145/3528416.3530232 (cit. on pp. 13, 14).

[19] Ayesha Siddique and Khaza Anuarul Hoque. «Improving Reliability of Spiking Neural Networks through Fault Aware Threshold Voltage Optimization». In: (2023), pp. 1–6. DOI: 10.23919/DATE56975.2023.10136999 (cit. on p. 13).

[20] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. «SoftSNN: low-cost fault tolerance for spiking neural network accelerators under soft errors». In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 151–156. ISBN: 9781450391429. DOI: 10.1145/3489517.3530657. URL: https://doi.org/10.1145/3489517.3530657 (cit. on p. 14).

[21] Guangba Yu, Gou Tan, Haojia Huang, Zhenyu Zhang, Pengfei Chen, Roberto Natella, Zibin Zheng, and Michael R. Lyu. «A Survey on Failure Analysis and Fault Injection in AI Systems». In: *ACM Trans. Softw. Eng. Methodol.* (May 2025). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3732777. URL: https://doi.org/10.1145/3732777 (cit. on p. 19).

[22] Garrick Orchard, Ajinkya Jayawant, Gregory K. Cohen, and Nitish Thakor. «Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades». In: *Frontiers in Neuroscience* Volume 9 - 2015 (2015). ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00437. URL: https://www.frontiersin.

org/journals/neuroscience/articles/10.3389/fnins.2015.00437 (cit. on pp. 23, 24).

[23] Li-Ye Niu, Ying Wei, Yue Liu, Jun-Yu Long, and Wen-Bo Liu. «Spatiotemporal Backpropagation based on Channel Reward for Training High-Precision Spiking Neural Network». In: *Signal, Image and Video Processing* 17 (May 2023), pp. 1–10. DOI: 10.1007/s11760-023-02569-0 (cit. on p. 24).

[24] Sören Becker, Marcel Ackermann, Sebastian Lapuschkin, Klaus Müller, and Wojciech Samek. «Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals». In: *ArXiv* abs/1807.03418 (2018). URL: https://api.semanticscholar.org/CorpusID:267800116 (cit. on p. 25).

[25] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. «The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks». In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: 10.1109/TNNLS.2020.3044364 (cit. on p. 26).