POLITECNICO DI TORINO

Master's degree in Data Science and Engineering

Master's Degree Thesis

Are spiking neural networks resilient to internal faults? A comparison with other neural network models and an analysis of possible solutions to increase resilience

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Dr. Alessio CARPEGNA

Candidate

Francesco GRANDI

October 2024

Abstract

Spiking neural networks (SNNs) are a new type of computational model whose potential is worth examining due to their similarity to the biological brain, of which AI researchers hope to harness the power and capabilities. These networks achieve comparable performances and accuracy to traditional neural networks in temporal data-related tasks while employing fewer resources. Maintaining the comparison with the biological brain, a question arises spontaneously: Are spiking neural networks able to retain the biological brain capability of resiliency, allowing it to remain functional even when damaged? The question is crucial when implementing artificial neural networks since they play a role in many aspects of our daily lives, including safety-critical areas like self-driving cars, disease detection, and more. Therefore, it is essential to assess their resilience to degradation, which frequently occurs throughout a typical software life cycle.

The main intent of this work is to compare the reliability of more traditional Artificial Neural Networks (ANNs) with Spiking Neural Networks. The methodology employed is based on conducting fault-injection campaigns. These processes entail deliberately introducing faults into the target model to evaluate its performance and robustness under faulty conditions. In particular, metrics such as accuracy and class probability on different datasets assess fault-caused damages on ANNs. In addition, a comparison with the most traditional types of Artificial Neural Networks is provided.

Acknowledgements

A heartfelt thank you goes to my parents, who worked alongside me to reach this goal, supporting me in every possible way. My grandmother Adele was always there when I needed her, and for this, I am deeply grateful.

I also want to extend my sincere thanks to my friends Andrea and Alice, who have known me for a long time and kept me great company during the days spent in study rooms. Mattia, Matteo, and Sam helped me unwind and free my mind from university worries on the weekends, and I thank them for that. A special thanks to Francesca for all the shopping we did together and Lorenzo for the thousands of trips we had together.

With Davide and Louiss, I shared the final stretch of this journey—moments I will certainly miss. I also thank Alessio Carpegna for his time and collaboration on this project, and Prof. Stefano Di Carlo for allowing me to work with his research group.

Last but not least, I thank my siblings, Andrea, Ilaria, and Elena, along with all my friends from Lucca and Torino, for their patience and support throughout these years.

Table of Contents

List of Figures

Chapter 1

Introduction

Data analysis, object recognition, and audio classification are a few of the many fields where Deep Neural Networks (DNNs) techniques are used to perform tasks that only a few years ago were manageable solely by humans or machines under human supervision. Nonetheless, as technology advanced and computational power became more and more available, Neural networks were able to perform more and more complex tasks. One of the most significant examples is Tesla's autonomous driving system, known as Autopilot, which leverages machine learning extensively. Other everyday-life examples are Netflix and Amazon machine learning systems used to personalize user content recommendations. Finally, ChatGPT-3, a Neural language model, has fundamentally changed the user experience of researching and gathering information. These computational models have an architecture inspired by the biological brain and take advantage of the enormous data available nowadays. While exploring ANNs potential, the attempts to mimic a biological brain were intensified, resulting in an increasing the number of neurons and complexity of the networks. For instance, Recurrent Neural Networks (RNNs) are ANNs attempting to work with temporal data, akin to the type of information processed by biological organisms in real-world scenarios. Likewise, Convolutional neural networks (CNNs), engineered

to classify images, were inspired by biological processes: The connectivity pattern between neurons took inspiration from the organization of the animal visual cortex [1]. The SNNs are therefore a mandatory step towards the perfection of ANNs, since they mimic the biological brain, transforming inputs in asynchronous events through action potentials or spikes, similar to how the neurons communicate. This capability makes them ideal for real-time processing of dynamic, time-sensitive data, particularly in edge-computing scenarios where energy and computational power are constrained. Some examples are IoT sensors deployed in home appliances or production sensors used to detect issues during machine operations. However, to deploy SNNs in safety-critical systems, it is essential to evaluate their reliability, as erroneous decisions could have serious consequences. One of the most effective methods for assessing the robustness of SNNs is to conduct extensive fault-injection campaigns. This process involves intentionally introducing faults before and during the system's inference operations and comparing the results to fault-free execution. This approach helps determine the impact of faults on the system performance, identify potential vulnerabilities, and improve the overall reliability of SNN applications in critical settings.

1.1 Workflow

As stated before, the objective of this thesis is to compare the fault resiliency of SNNs to that of traditional DNNs. In the process, two main frameworks were employed:

- snnTorch: designed to be intuitively used with PyTorch, it allows inserting special neuron layers into the networks allowing it to work with potentials and spikes.

- ANNFI: A fault injection tool operating on ResNet networks through both stuck-at and bit-flip faults.

The choice of these tools was straightforward since the first is widely used and well-documented. The latter was developed internally within Politecnico di Torino, making it very easy to obtain assistance. From these initial points, the work was divided into four main phases:

1. Choosing the type of Computational model to test: Both the number of Layers and type of layers were a crucial choice since the simulation time, and complexity of the experiments were directly related to them.

2. Adapting ANNFI framework to work with Spiking Neural Network because Spiking neural networks have a different set of parameters.

3. Simulation on Linux server and result analysis: Since every simulation lasted several hours, it was necessary to employ Linux servers equipped with NVIDIA GPUs. The results obtained were then thoroughly examined with parameters-per-layer granularity.

1.2 Why Python

The choice of Python programming language was straightforward. Python language is known for its simplicity and consistency. The Python code is concise and readable, which simplifies the presentation process. It is widely used in Machine learning since it implements PyTorch and TensorFlow, two of the main frameworks in the field. In addition, python implements libraries such as Numpy and Matplotlib, that come in handy for data engineering and data Analysis. These two steps were essential for elaborating faults-injection results. Finally, ANNFI and snnTorch were built in Python, therefore, to avoid wasting time rewriting the same code in different languages, Python was an obliged choice.

Chapter 2

Background

This chapter is a brief introduction to the main topics covered by the thesis. A concise explanation of the functioning of machine learning algorithms implemented is provided. In particular, it presents a brief explanation of the main theory behind Convolutional and Feed Forward Neural networks. Afterward, it is well explained why studying SNNs' potential could prove very useful for future machine learning developments while considering its pros and cons. It then follows an explanation of the differences between Artificial Neural Networks and spiking neural networks. Finally, the main elements constituting SNNs are outlined.

2.1 Traditional Neural Networks

Neural Networks are a kind of Artificial Intelligence able to perform complex tasks. They are one of the more complex A.I. models and each recent DNN has at least thousands of neurons if not billions. To tune these algorithms and teach them to perform the task desired, it is necessary to have huge amounts of both data and computational power.

Modern research in the field started in 1943, when Warren McCulloch and

Walter Pitts published a pioneering work outlining a threshold linear combiner. It is the first example of an Artificial neuron, composed of multiple binary inputs and a single binary output. An appropriate number of such elements, connected to form a network, is capable of computing simple Boolean functions [5]. Nonetheless, it wasn't until 1958, that psychologist Frank Rosenblatt invented the Perceptron, the first implemented artificial neural network [6]. To provide a brief explanation of their functioning, the case of a slightly more complex design is considered: the Multilayer Perception (MLP).

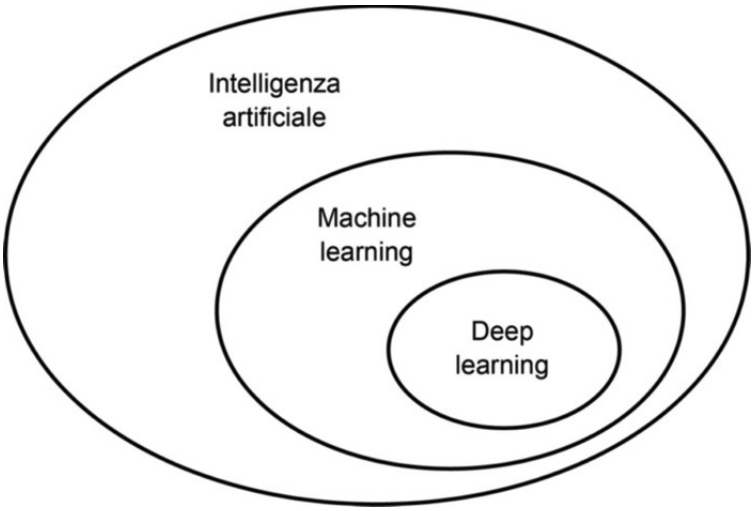


Figure 2.1: Artificial Intelligence hierarchy

2.1.1 Multilayer perception and Feed Forward Neural Networks

A multilayer Perceptron is a kind of feedforward Neural Network (FNNs) and is one of the simplest examples of ANN. It comprises one input and one output layer and one or more hidden layers between them. These hidden layers are formed by neurons, working similarly to a perception. It is important to underline that ANNs are fully connected networks, where every neuron of a layer is connected to every other neuron of the previous and following layer.

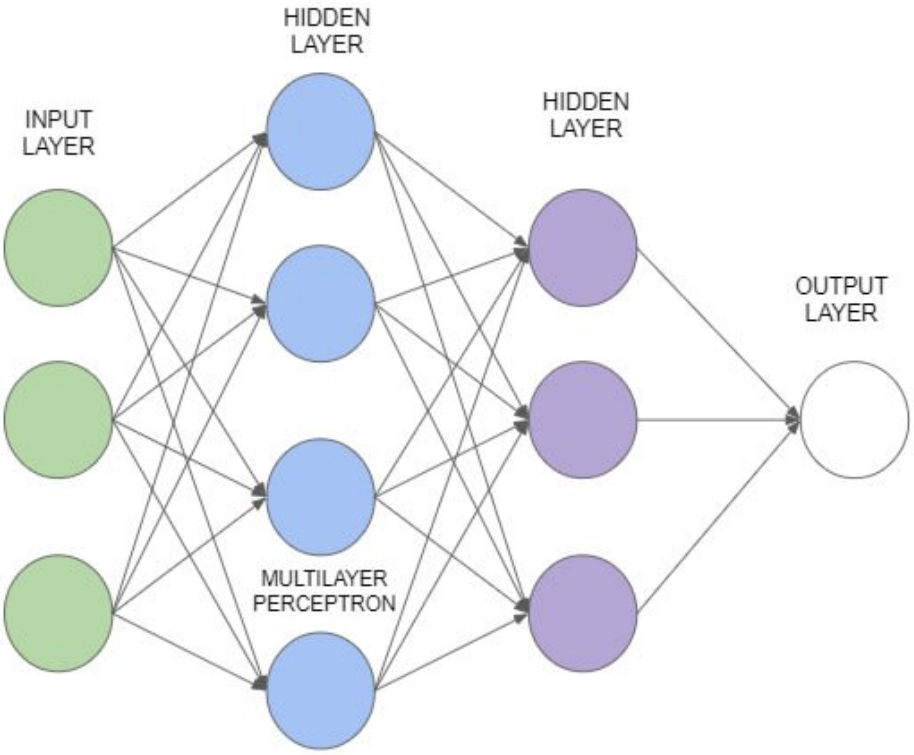


Figure 2.2: Example of multilayer perceptron

Each input layer corresponds to a feature of any single sample of data. Its purpose is to send to each next-layer neuron the input feature, scaled by a certain

weight. Each internal neuron layer sums up the new scaled inputs while adding a bias and an element of non-linearity by introducing an activation function (a non-linear function). Afterward, each neuron sends to each next-level neuron the newly generated information.

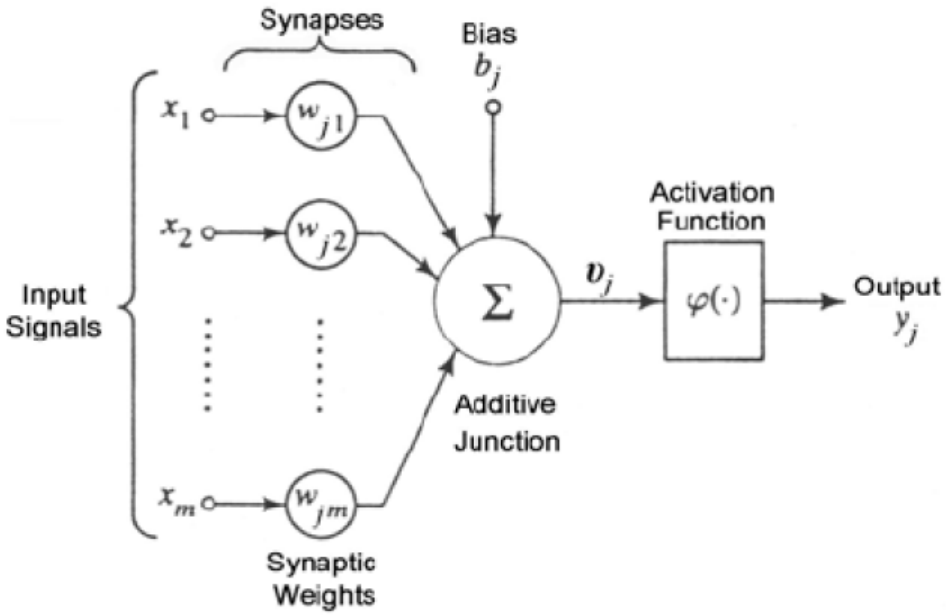


Figure 2.3: Scheme of a LIF neuron mechanism

This mechanism called forward propagation, goes on until the output layer is reached. Here, after summing up the previous layer-generated inputs and adding bias, a function is used to convert the information into probabilities, each corresponding to a class. In general, each DNN has two main phases, training and inference. The process described before is present in both phases, but while training there is one more step needed. The values of the weights, present throughout the network, need to be set since they are randomly determined as the network is defined at the beginning. This happens through backpropagation, which takes place by computing the loss, a measure computed, for example, as the squared difference between the final probabilities obtained and the actual class. Afterward,

while the Loss is backpropagated through the network, the weights are adjusted. Finally, it is important to underline that the process described above is meant only to outline the functioning of a neural network, which is regulated by complex equations with many factors and variables to take into account.

2.1.2 Convolutional Neural Network

This algorithm represents the state-of-the-art solution for a wide range of computer vision tasks, specifically those aimed at classifying one or more objects within a single 2D image. Thereby, CNNs are widely used in image classification and object detection applications due to their outstanding results, often matching or surpassing human accuracy. A significant limitation of previously mentioned approaches was their inability to ensure shift, scale, and distortion invariance of input images. This may be partially addressed through image pre-processing, which requires considerable effort for each image. CNNs, however, adopt a fundamentally different method by learning features from sets of labeled images (training sets) to classify various inputs. The structure of CNNs is quite similar to ordinary neural networks, with neurons grouped into layers to extract informative features from inputs. However, CNNs utilize smarter constraints on layer shapes. While a simple neural network can perform image classification, it is impractical due to the vast number of parameters needed, resulting in high training time and memory requirements. For instance, an RGB image of 256 $\times$ 256 pixels would require an input layer with approximately 300,000 neurons, potentially leading to millions in the hidden layers. To address this issue, concepts from biology have been incorporated. Studies on the cat's visual cortex demonstrate that only small regions of neurons are interconnected. This principle was first applied to neural networks by Kunihiko Fukushima [1] with the development of the initial CNN model, and later enhanced by Yann LeCun [7] in 1998 with the trainable CNN

model, LeNet5. Key features introduced in these works include local receptive fields, shared weights, and sub-sampling. These features significantly reduce the number of parameters to be learned, as each neuron is connected only to a small region of the previous layer, not all previous units.

As already underlined CNNs have a more complex structure that is used to filter and select the most characteristic pattern of an image. In the following paragraphs, a brief explanation of each layer is given.

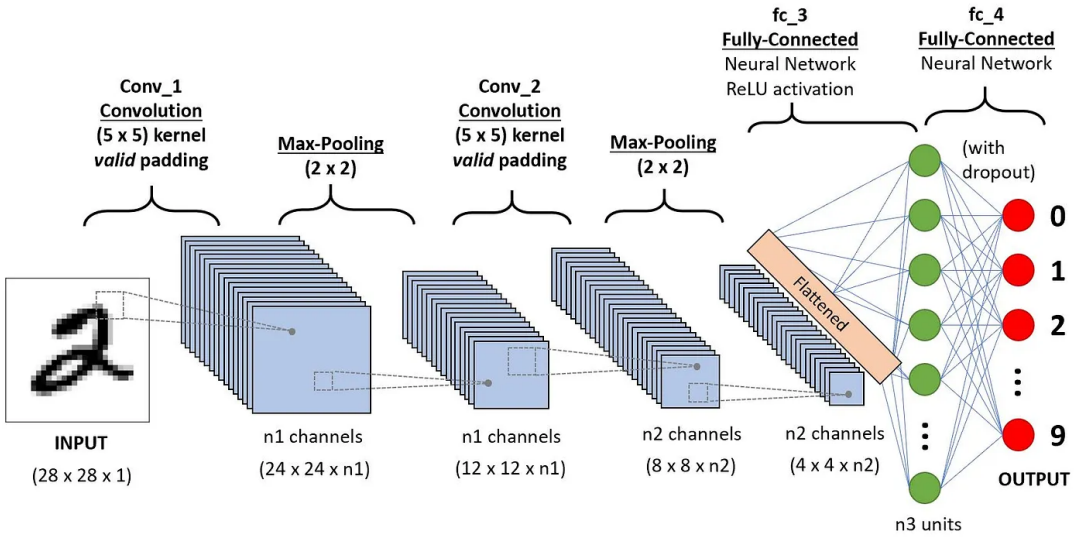


Figure 2.4: Architecture of the CNNs applied to digit recognition

Convolutional layers

This is the foundational component of a CNN. As the name implies, its primary mathematical operation is convolution, which involves applying a sliding window function to a matrix of pixels representing an image. This sliding function, known as a kernel or filter, can be used interchangeably. In the convolutional layer, multiple filters of the same size are applied. Each filter detects specific patterns in the image, such as the curves of digits, edges, or the overall shape. Simply put, in the

convolutional layer, we use small grids (filters or kernels) that move across the image. Each small grid searches for specific patterns like lines, curves, or shapes. As it moves over the image, it creates a new grid that highlights the locations of these patterns. For instance, one filter might be used to find straight lines, while another might identify curves. By using multiple filters, the CNN can recognize a wide variety of patterns within the image. Every Convolutional layer receives an input image, called the input feature Map, while in output produces the output feature map. They are represented in Tensor form, with dimensions ($c \times h \times w$), respectively the number of channels, height, and width. The hyperparameters that determine the shape of the convolutional layer are depth, stride, and padding. Depth refers to the number of filters applied, each learning to detect specific features. The stride indicates the number of pixels by which the window moves; for example, a stride of 1 means the window moves one pixel at a time. Finally, padding specifies the number of zero-valued pixels added around the border of the input.

Pooling layers

The purpose of the pooling layer is to extract the most important features from the convoluted matrix. This is achieved by performing aggregation operations that reduce the dimensions of the input feature map, thereby decreasing the memory required during network training. For example, max pooling selects the highest value from the feature map while sum pooling aggregates all the values in the feature map, and average pooling calculates the mean of all the values in the feature map.

Fully connected layers

In addition to the new layers mentioned above, fully connected layers are implemented in CNNs as well. These layers are the final ones and their inputs correspond to the flattened one-dimensional matrix produced by the last pooling layer. It is important to underline that each convolutional and fully connected layer has a ReLU activation function that helps the algorithm learn non-linear relationships between the image and the extracted features.

2.2 From ANNs to SNNs

Why SNNs?

For how incredible the results achieved by CNNs and FNN are, many problems may arise during their implementation. For example, the **computational power** needed to run top-performing deep learning models has increased ten times each year from 2012 to 2019 [8, 9]. Similarly, data generation is rising at an exponential rate. OpenAI's ChatGPT language model, GPT-3, which has 175 billion learnable parameters, is estimated to require about 190,000 kWh [10, 12, 13] to train. In contrast, our brains operate on just 12-20 W of power, handling numerous sensory inputs while maintaining essential involuntary biological functions. In this context, SNNs are very useful since they operate with a sparse process and need less energy. They utilize an event-driven mechanism where neurons only fire or 'spike' in response to specific stimuli. This sparsity of activity means that at any given time, only a small fraction of neurons are active, drastically reducing energy consumption.

In addition, the intrinsic capability of Spiking Neural Networks (SNNs) to handle temporal information grants them a significant advantage in tasks involving sequences or time-based data. This advantage is crucial because biological brains operate in dynamic environments, continuously receiving new information from

our senses and internal organs. In contrast, traditional Artificial Neural Networks, including those used in Large Language Models (LLMs), often need supplementary mechanisms like recurrent neural networks (RNNs) or long short-term memory (LSTM) units to manage temporal dependencies. SNNs, however, integrate time directly into their computational processes, enabling them to model sequences and temporal patterns more efficiently. The following paragraphs introduce the most important elements of SNNs focusing on the elements implemented during the project. Few interesting article on the SNN topic are [11], [2], [3], [4]

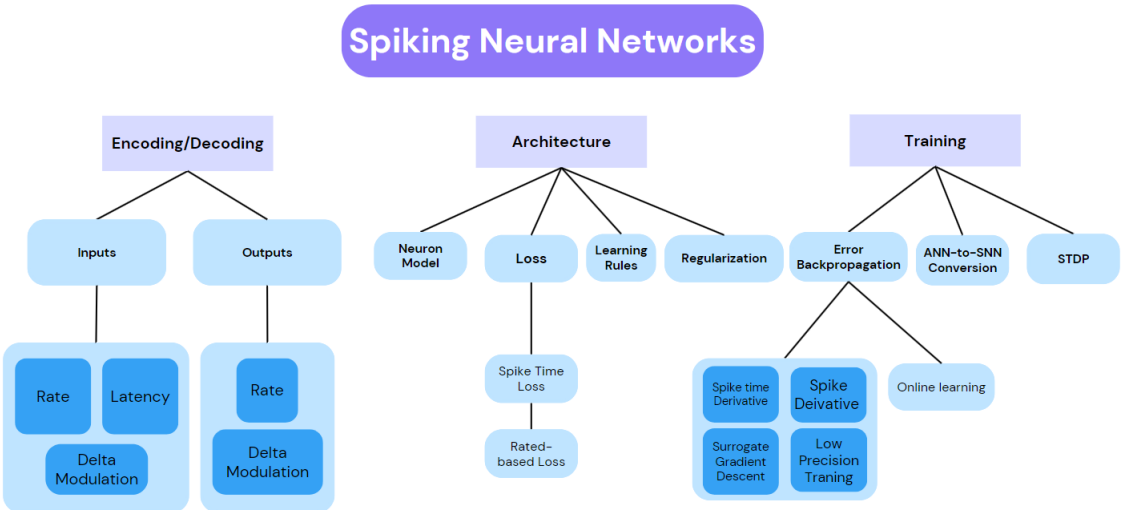


Figure 2.5: Diagram of the SNNs main characteristics

2.2.1 Neuron model

LIF neurons

One of the main topics of the whole spiking neural network subject is the neuron mechanisms. In particular, leaky integrate-and-fire (LIF) neurons are the most used and the most energy-efficient kind of SNNs neuron. Their mechanism is based on receiving input spikes (a.k.a electrical impulses). Each increases the membrane potential $U(t)$ according to a scaling factor. Afterward, a weighted of the spikes is

computed. A spike is generated and propagated to subsequent neurons if a certain electrical threshold θ is reached.

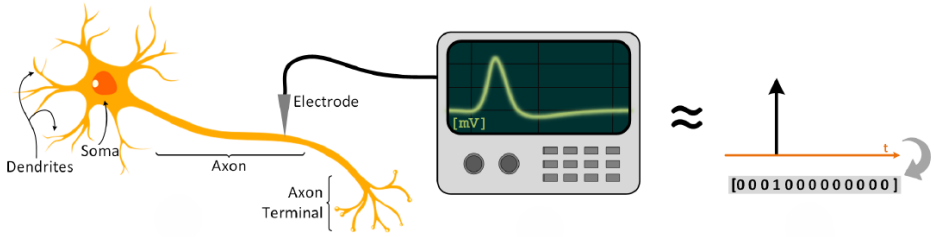


Figure 2.6: From neurons to spikes [24]

It is important to emphasize that the capability of SNNs to manage sequential and temporal data can be explained for the most part by the mentioned mechanism: The potential reached through spikes' contributions is constantly decreased by a factor until it becomes null. Since only one spike contribution is probably insufficient to reach the threshold, the timing and frequency of various spikes are essential to sustain $U(t)$ until the threshold is reached. Incoming spikes are given by the input data of the whole network; if it is temporal-sequential data, the spiking neural network will interpret it as such incorporating the time dynamics.

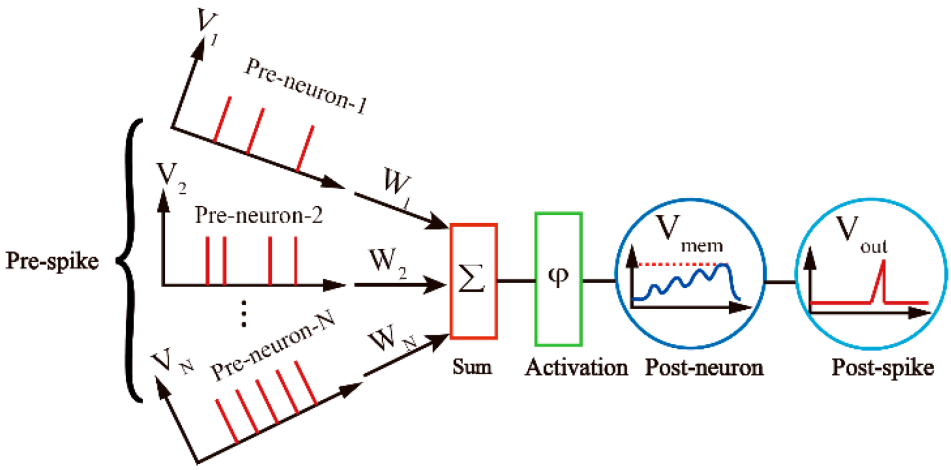


Figure 2.7: Scheme of a LIF neuron mechanism

This neuron process resembles the ones observed by Louis Lapicque in his experiments by stimulating a frog nerve through an electrical current. He was able to understand when the leg frog would have twitched by varying the amplitude and intensity of it. He understood that the nerve mechanism resembled coarsely a low-pass filter circuit consisting of a resistor R and a capacitor C [**?**, 14]. Physiologically this holds up: the capacitance is due to the insulating lipid bilayer that constitutes the neuron's membrane. The resistance is a result of gated ion channels that open and close, regulating the diffusion of charge carriers across the membrane.

To explain this mechanism well few formulas are derived from the standard RC-circuit equation:

$$\tau \frac{dU(t)}{dt} = -U(t) + I_{in}(t)R \tag{2.1}$$

where $I_{in}(t)$ is the input current to the neuron and $\tau = RC$ is the time constant of the circuit. Typical values of τ fall on the order of 1-100 milliseconds. To adapt this time-varying solution for a sequence-based neural network, the forward Euler method is employed in the simplest scenario to obtain an approximate solution to Equation 2.1 [16]:

$$U[t] = \beta U[t-1] + (1-\beta) I_{in}[t] \tag{2.2}$$

$\beta = e^{-\frac{1}{\tau}}$ is the decay rate of the potential generated by the input spikes. The smaller is τ the more the potential decreases with time and the more the new input current is important to the present state. To give more freedom to the model to learn, the scaling factor of the input current $1-\beta$ is changed to a learnable parameter W. Finally, the threshold reset factor is added to 2.2:

$$U[t] = \beta U[t-1] + WX[t] - S_{out}[t-1]\theta \tag{2.3}$$

where

$$S_{out}[t] = \begin{cases} 1 & \text{if } U[t] > \theta \\ 0 & \text{otherwise} \end{cases}$$

The above equation exemplifies the 'reset-by-subtraction' (or soft reset) mechanism: when the threshold is reached at $t-1$ a quantity equal to the maximum potential reachable θ is subtracted, resetting U to zero.

Alternative to leaky neurons

To offer a more complete overview, a few other types of neurons are introduced:

- **Integrate-and-Fire (IF)**: The leakage mechanism is removed; $\beta = 1$ in Equation 2.3.

- **Current-based (CuBa)**: These incorporate synaptic conductance variation into leaky integrate and fire neurons. The potential has continuous values rather than experiencing discontinuous jumps in response to incoming spikes. They are more realistic but do not result in better performances.

- **Higher-complexity neuroscience-inspired models**: A large variety of more detailed neuron models are out there. These account for biophysical realism and/or morphological details not represented in simple leaky integrators. The most renowned models include the Hodgkin-Huxley model [15]

- **Recurrent Neuron**: The output spikes of a neuron are routed back to the input and can be implemented in different ways; i) one-to-one recurrence, where each neuron routes its own spike to itself, or ii) all-to-all recurrence, where the output spikes of a full layer are weighted and summed before being fed back to the full layer.

- **Kernel-based Models**: Also known as the spike-response model, where a pre-defined kernel (such as the 'alpha function') is convolved with input spikes. The option to define the kernel to be any shape offers significant flexibility.
- **Deep learning inspired spiking neurons**: Rather than drawing upon neuroscience, it is just as possible to start with primitives from deep learning and apply spiking thresholds. This helps with extending the short-term capacity of basic recurrent neurons.

The model's intent should ultimately determine the choice of the neuron to implement. Energy efficiency calls for Leaky neurons while for accuracy is better to implement Recurrent neurons.

2.2.2 Encoding and spikes

Input encoding

Just as the biological brain, multiple diverse inputs can be fed to the Spiking Neural Network, may they be images or audio and signals. Of course, the SNNs neurons do not convert autonomously inputs in spikes. Therefore, together with the canonical data preprocessing, an additional step may be needed to convert input data into spikes.

The main coding techniques are:

1. **Rate coding**: translates the intensity of an input into a firing rate or the number of spikes. For instance, if a neural network is trained using a black-and-white image dataset, the brighter a pixel is, the more spikes it generates. One of the advantages of this method is to have a high error tolerance since if a spike goes missing there are many more to signal the intensity of the input.
2. **Latency (or temporal) coding**: transforms input intensity into the timing of spikes. In this case, it is more important when the spikes are generated

rather than their quantity. Even if the method is more prone to noise is less energy expensive: generating and communicating fewer spikes means less dynamic power dissipation in a tailored hardware.

3. **Delta modulation**: converts changes in input intensity over time into spikes, remaining silent otherwise. It is based on the notion that many receptive organs in the biological realm are adapted to perceive rapid changes. Therefore, introducing this technique allows for neural networks more similar to the biological brain.

In general, Input data to an SNN does not necessarily have to be encoded into spikes. It is acceptable to use continuous values as input, similar to how the perception of light starts with a continuous stream of photons hitting our photoreceptor cells.

In the case of this project, none of the above techniques has been employed since biological realism was not required to test the fault tolerance of Neural Networks. Instead, through *Tonic* Library it was possible to download the desired dataset in the form of events and transform each dataset entry in a few frames, depicting the data through time. For each timestep, a batch of frames was passed to the network to train it. In the end, accuracy was computed over all timesteps.

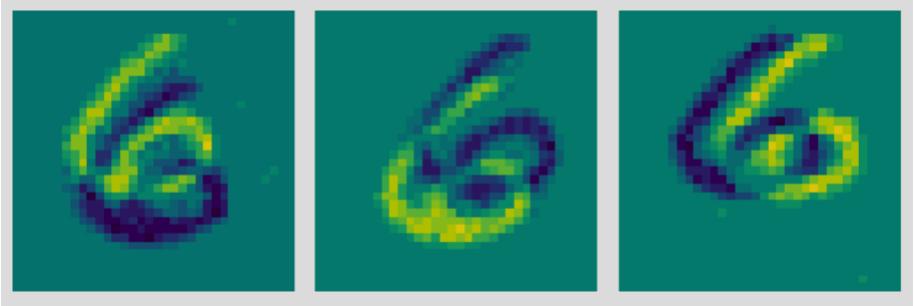


Figure 2.8: Few frames of an element of datasets NMNIST

Output Decoding

In addition to input encoding, a way to interpret the output spikes must be chosen. In fact, how can we infer the quantity and the timing of the spikes that indicate the classification of one class rather than another?

Also, in this case, the choice depends on the network's main purpose. If energy optimization is prioritized, then Latency coding is the choice. If error tolerance is needed then Rate coding should be implemented. In addition to the above methods, population coding can be implemented together with them. In this case, multiple neurons correspond to a class. It must be underlined that the output coding does not depend on the choice of input coding: During training, the algorithm adapts both to inputs and output encoding as they determine the loss and therefore the adjustments of weights. Regarding this project, rate output coding has been implemented through *snntorch.functional.acc.accuracy_rate* since it was not necessary to simulate biological realism.

2.2.3 Training

An essential part of ANNs functioning is how the network is trained. The same stands for SNNs.

- **Shadow training**: A ANN is trained and then converted into an SNN by interpreting the activations as either firing rates or spike timings
- **Backpropagation using spikes**: The spiking neural network (SNN) is trained directly using error backpropagation, usually through time, akin to the training process for sequential models.
- **Local learning rules**: Weight updates are determined by signals that are specific to the local region and timing of the weight, instead of using a global signal as in standard error backpropagation

Backpropagation using spikes was implemented at this juncture. Therefore, all SNNs implemented were trained as such from the beginning and without local learning rules. Converting trained ANNs to SNNs implies abiding by a number of restrictions for any of the currently available conversion methods to work properly [17] and at the same time it is an underexplored area [16]. In addition, since this thesis aims to be as general as possible without focusing on a particular situation, local rules were not chosen to train SNNs as they are not very common as well as not the main method to train ANNs in general.

Backpropagation

Backpropagation is a gradient estimation method used to train neural network models. The gradient estimate is used by the optimization algorithm to compute the network parameter updates. It consists of computing the gradient of the loss with respect to the weights, which are the parameters to update. Finding the zero of the gradient is equivalent to finding the minimum of Loss. To do so, the gradient is computed through the chain rule, which is a method used to differentiate composite functions. The whole process is iterated backward from the last layer to avoid redundant calculations of intermediate terms.

In the case of SNNs, the loss regarding only one neuron can be computed as:

$$\frac{d\mathcal{L}}{dt} = |W_{out}S_{out} - y| \tag{2.4}$$

if we consider the gradient (which is one-dimensional in this case since we are dealing with only one neuron) we obtain:

$$\frac{d\mathcal{L}}{dW_{out}} = S_{out} \tag{2.5}$$

while in order to compute W_{in} we apply the chain rule starting from the output spike S_{out} going backward until weights before the neuron W_{in}:

$$\frac{d\mathcal{L}}{dW_{in}} = \frac{d\mathcal{L}}{dS_{out}} \frac{dS_{out}}{dU} \frac{dU}{dW_{in}} \tag{2.6}$$

This mechanism, fundamental for the functioning of a Neural Network, may present many problems. One of the most important is the dead neuron problem caused by the fact that the analytical solution of (2.6) results in a gradient that does not enable learning. This derives from the the term $\frac{dS_{out}}{dU}$ has only two solution: 0 or ∞. Thereby, no weight can be found to optimize (2.6) when nearly always 0, effectively killing the neuron, by not allowing it to contribute to the overall learning of the network.

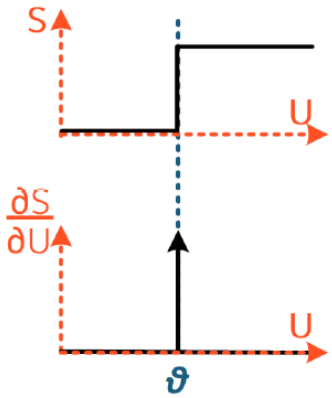


Figure 2.9: Plot of the S value, which is 0 until the spike is fired, plotted against the potential. The latter allows for the spike firing only when θ is reached.

From the above image, it can be seen that $\frac{dS_{out}}{dU}$ is 0 if $U \neq 0$ and ∞ if $U = 0$.

Surrogate gradient

To solve this problem a mathematical escamotage is implemented: The Surrogate gradient. It involves leaving the forward pass unchanged while substituting a continuous function during backpropagation to $S(U)$. One example of a continuous function that can be implemented in this context is the threshold-shifted sigmoid function:

$$\sigma = \frac{1}{1 + e^{\theta - U}} \tag{2.7}$$

and its derivative is:

$$\frac{d\tilde{S}}{dU} = \sigma' = \frac{e^{\theta - U}}{(1 + e^{\theta - U})^2} \tag{2.8}$$

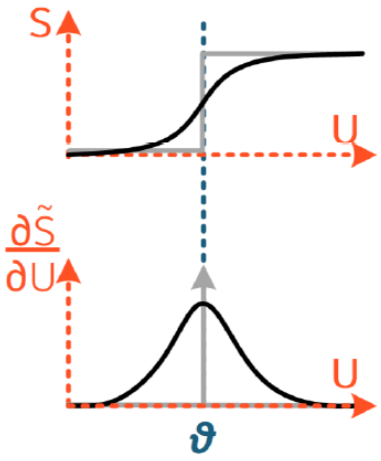


Figure 2.10: Plot shifted sigmoid function and its derivates

In this way, it was possible to bypass the dead neuron problem. Nonetheless, for backpropagation to work a neuron needs to fire at least once. A dead neuron is also a neuron that fails to activate and, as a result, does not impact the loss function as its gradient is always 0. Consequently, the weights associated with this neuron do not influence the credit assignment process. This prevents the neuron from learning to activate in the future.

2.3 Fault-injection

Software and hardware testing is a mandatory phase in the development of any cyber-physical system that will be ultimately released on the market. This is

particularly relevant in safety-critical environments, where SNNs, like many ANNs currently are, might be deployed. Safety-critical applications are governed by numerous international standards that establish a framework of rules and metrics developers must adhere to in their work. For example, ISO 26262 is a standard specifically created to outline testing procedures and regulations for the automotive industry, aiming to prevent fatalities resulting from both hardware and software failures. In this context fault injection is one of the most used tools for detecting software bugs and errors. In the next sections, some coordinates will be given to navigate better the vast world of software testing.

2.3.1 Software implemented fault injection

The first attempts at fault injection go back to 1970 [18]. While mostly related to hardware testing, they were based on the same principle of modern fault injection, namely exploring the space of possible fault, to understand better hardware and asses its reliability. It involved nothing more than shorting connections on circuit boards and observing the effect on the system (bridging faults). Nowadays, as already mentioned, fault injection is a common practice and usually it's designed keeping in mind a few variables:

- **Type**: What type of in injection is performed. To mention a few, it can be bitwise, stuck-to-value, delayed, or simply ignored some parameters or functions.
- **Timing**: The choice of the fault injection is very important as the impact of the algorithm might change drastically.
- **Location in the software**: where should be in the system? For example, faults can be located within the systems/subsystems/functions or in the link/connection between systems.

These parameters combined originate the fault space, intended as the set of all possible combinations of faults. Its dimension increases exponentially as the complexity of the tested system grows. Therefore, it is crucial to choose wisely the before-mentioned variables as it would be impossible to examine all fault-injection space given its dimension. This is directly reflected in this project: only a small subset was selected from a complete list of all possible faults. Otherwise, days-long simulations would have been necessary instead of hours-long ones.

In the context of this project, an exclusively Software Implemented Fault Injection (SWIFI) is implemented. This implies that no hardware factor was taken into account to test SNNs. SWIFIs are mainly categorized into two types:

- **Compile-time Injection**: Compile-time fault injection is the simplest type of fault injection because allows the injection of fault at compile-time, by changing the code of the software module under test. One of the most widespread example is called Mutation test, which changes code statements to insert data or operation perturbations.

- **Runtime Injection**: Run time fault injection is more complex since requires a software trigger that explicitly injects a fault into the source code. However, it is more powerful, compared with compile-time injection, since allows the simulation of either hardware or software faults. It takes place during the execution of the program.

In the context of this work, both methods were employed. The fault injection framework ANNFI creates a fault list which is a randomly chosen subset of the fault space. Then, the faults are injected and the inference algorithm is run (compile-time injection). Nonetheless, runtime injecting was a choice made necessary by membrane potential and spike faults injection: they are generated during the inference process and passed between layers. Since they are meaningful only in the

iteration during which they are generated it would have been very time inefficient to save them, inject the faults, and then feed them again to SNNs at the right moment during compile-time. Thereby, the faults were injected in runtime during the inference phase as they were computed.

Chapter 3

Proposed Approach

In this chapter, we outline the methodology used for fault injection while evaluating its pros and cons. As already mentioned, the main objective of this thesis is to conduct a reliability analysis focused exclusively on spiking convolutional neural networks (CSNNs) and spiking feed-forward neural networks (FSNNs), assessing their fault tolerance during the inference phase by identifying deviations from the correct results. To accomplish this, a simulation-based fault injection method is proposed, where faults are intentionally introduced into the system to evaluate the response of the networks, also in comparison with standard CNNs and FNNs.

3.1 ANNFI

ANNFI is a framework that serves as the starting point for fault-injecting on SNNs. The modules already present were adapted to the new model without changing their logic. The modified ANNFI is mainly composed of two parts:

- Fault List Generator (FLM)
- Fault Injection Manager (FIM)

Fault List Generator

The FLM is used to generate the faults, based on the user-requested type of parameters to inject. For each, according to the network architecture, all the possible instances of the selected type are considered. Nonetheless, only a few are selected, each characterized by layer and tensor indexes. In addition, to perform a single-bit injection, the position of the bit is randomly chosen out of 32. The possible parameters to choose from for fault injecting can be divided into three categories:

- **Parameters present both in ANNs and SNNs:**
  - Weight
  - Bias
- **Parameters present only in SNNs:**
  - Beta
  - Threshold
- **Parameters present only in SNNs injected in runtime:**
  - Spikes
  - Membrane potential

Even if more parameters can be considered, the ones selected are the most significant in the dynamics of SNNs and ANNs. It is worth mentioning that the sample of parameters selected is chosen according to statistical fault injections (SFIs) formulas, which are currently regarded as a valid method to estimate a specific characteristic of a population of faults by observing only a sample. These measurements are based on an error margin ϵ, a confidence level t, and the

numerosity of the complete sets of faults N. The numerosity of the sample is computed as follows:

$$n = \frac{N}{1 + e^2 \frac{N-1}{t^2 p(1-p)}} \tag{3.1}$$

Where p is the probability of success. This term is present in the equation because SFI can be seen as repeated n trials of injection, with a p probability of success (a trial is a success when a fault becomes a critical failure). In this context, a conservative choice of p is 0.5 since it assumes the same probability of success or failure of the network. The resulting fault list is therefore more comprehensive. In addition, following [19] e and t have been set respectively to 1% and 99%.

Fault Injection Manager

The FIM's purpose is to run the clean and faulty campaign. A clean campaign is a cycle of inference over the dataset with a not-fault-injected network. A faulty campaign comprises many cycles over the dataset, each with the same ANNs but injected with a different fault. It is important to underline that the FLG, to generate the complete Fault List (FL), needs information regarding the shape and number of tensors representing spikes and membrane potential as they are difficult to deduce before the machine learning algorithm is deployed. Therefore, at the first iteration of the clean campaign, the information is passed to FLG to complete the FL. Once the FL is created and saved, FIM feeds the dataset to the ANNs selected for inference and saves golden values, namely the metric obtained from fault-free networks. Afterward, the fault is injected and FIM runs the ANNs in inference mode on the dataset. It does so one fault at a time, for the whole dataset, while saving the necessary metrics for data analysis. Finally, the fault-free network is restored and the following fault in the FL is inserted. The process repeats itself until there are no more faults in the list. This process is schematized in 3.2.

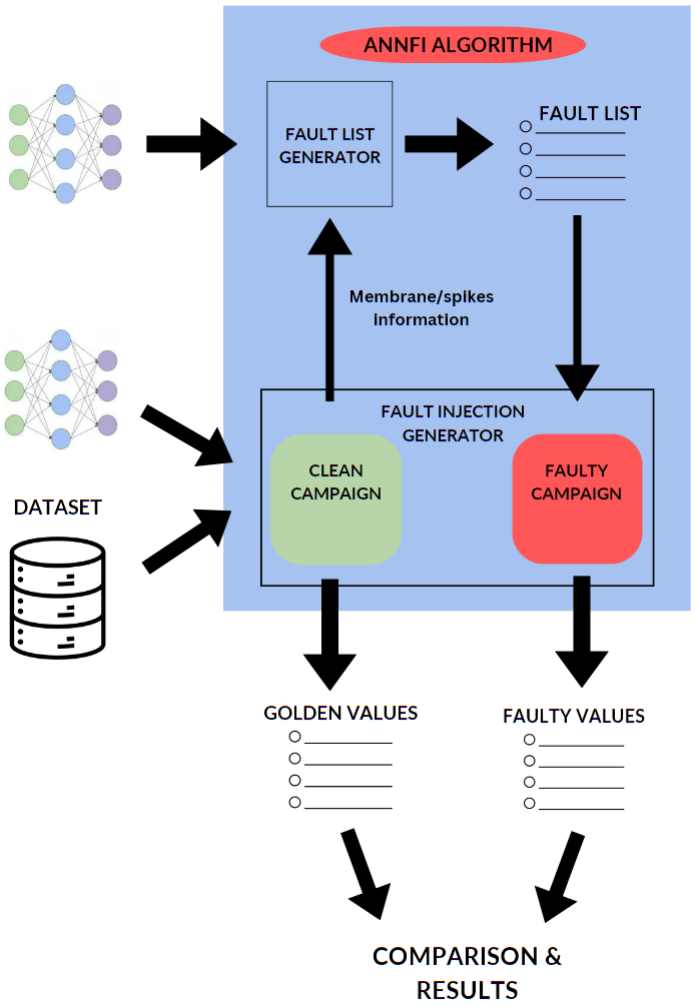


Figure 3.1: Modified ANNFI framework

3.2 Simulation design and simulation life-cycle

During the development of the Fault injection, many simulations were performed and the output was analyzed to verify that the data made sense in the context of SNNs fault injection. In particular, instruments such as heat maps proved very useful in quickly overviewing how the output data correlated. For instance, the masked faults percentage is likely negatively correlated with the delta values since

the more masked faults there are, the lower the value of deltas. This is because deltas are defined as the difference between golden accuracy and after-injection accuracy. In general values between ±0.50 and ±1 suggest a strong correlation while values between ±0.30 and ±0.49 indicate a moderate correlation. Instead, values between ±0.29 suggest a weak correlation. These considerations helped assess the correctness of outputs with thousands if not millions of entries each.

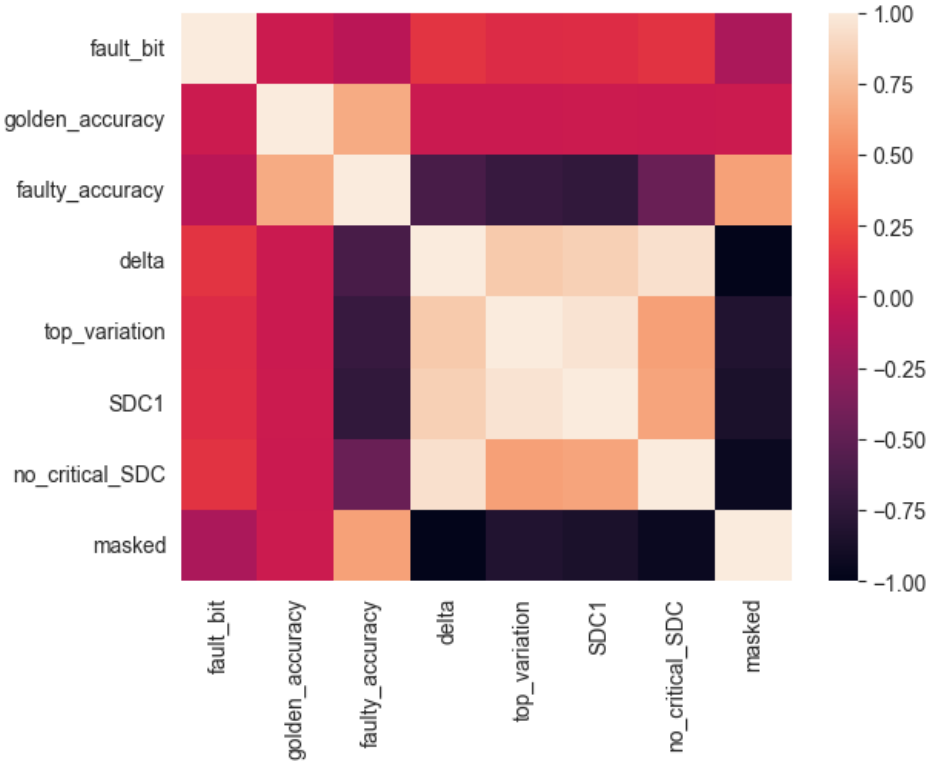


Figure 3.2: Example of a heatmap used during data analysis. The lighter the color the higher the correlation

After the simulation mechanism was complete the second part, centered around simulations and their analysis, started. In the process, a "life cycle" of simulation can be identified. It comprises a few phases, repeated always in the same order, which were the most present during the simulation process:

1. **Simulation set-up:** To perform a simulation, a few parameters must be set, both through the command line and by modifying the program. Among the most important: network to use, batch size, the flag for CUDA, and type parameters.

2. **Fault-injection:** The simulation itself, which might have lasted several hours; was crucial to choose wisely the parameters to inject since injecting all of them together would have been very time-consuming.

3. **Result Analysis:** Through Jupyter notebooks the results have been analyzed thoroughly implementing data engineering and using many kinds of plots (bar plots, heatmaps, regression lines, and similar). This step was fundamental as it set the next simulation setup, determining therefore the next cycle.

3.3 Type of faults

ANNFI framework supports the injection of two categories of faults, affecting synaptic connections (weight and biases) and neuron activity, such as membrane potential and spikes:

- **Stuck-at**: This technique allows to fix the value of the variable to 0 (suck-at-0) or 1 (suck-at-1) during inference time. This is particularly meaningful for SNN fault injection, as spike values are binary: A value equal to zero is equivalent to the absence of a spike in a time t and vice versa.

- **bit-flip**: It consists of flipping one of the 32 bits representing a value.

In the context of this project, Stuck-at fault injection has been the most used to asses network resilience. This is because bit-flipping a spike generates float values, not directly compatible with binary spikes. Therefore is more straightforward, and easy to evaluate stuck-at-type faults.

Chapter 4

Experimental Setup and results

This chapter describes the experimental setup, including the software and frameworks used. A detailed description of Datasets, neural network models, and the experiment setup is mandatory to guarantee the repeatability of experiments.

To begin, the dataset implemented for model tuning and testing is described. They represent two very different types of data and to obtain acceptable model performances an important part of the thesis was devoted to preprocessing and model tuning. In particular, the models considered are CNNs and FNNs and their respective spiking versions (CSNNs and FSNNs). Once the models were obtained, they were inserted into ANNFI, after the necessary changes to the framework.

4.1 Experimental Setup

As already mentioned, to launch a simulation few parameters must first be specified. Accordingly, ANNFI automatically selects the model's dataset. The following sections will thoroughly examine the above elements in this chapter.

4.1.1 Datasets

The choice of datasets has been made according to the preexisting work on the topic. Comparing results based on similar data decreases the variables to consider while conducting experiments on the topic. In addition, they are very different datasets as one is made of images of numbers, and the other is comprised of preprocessed sound signals. This diversity should help to make the final analysis more robust.

The two chosen datasets are:

- **Spiking Heidelberg Dataset (SHD):** It consists of approximately 10000 high-quality aligned studio recordings of spoken digits from 0 to 9 in both German and English language. Each element of the dataset is made of spikes in 700 input channels that were generated using Lauscher, an artificial cochlea model. Recordings exist of 12 distinct speakers two of which are only present in the test set [20].

- **Modified National Institute of Standards and Technology database (MNIST):** The MNIST is a database of handwritten digits with a training set of 60,000 examples and a test set of 10,000 examples. The digits, from 0 to 10, have been size-normalized and centered in a fixed-size image. The resulting images contain grey levels due to the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image [21].

- **Neuromorphic MNIST (N-MNIST):**: It is a spiking version of the original frame-based MNIST dataset and It consists of the same 60000 training and 10000 testing samples as the original database. It was created by capturing 28x28 8-bit grayscale images of handwritten digits using an ATIS event camera. The dataset consists of 10 classes, representing the digits 0 through 9 [22].

	SHD	NMNIST
CNN	70.79%	82.88%
CSNN	64.51%	87.64%
FNN	79.88%	95.96%
FSNN	82.40%	95.32%

Table 4.1: Models accuracies on datasets

4.1.2 Preprocessing

Neuromorphic datasets are arrays of zeros and ones representing spikes at a certain time t. Feeding this data format to traditional neural networks would have been very difficult. *Tonic* framework came in handy since it allowed for converting each event into frames or images, once the time window or the number of events to consider was specified.

The datasets mentioned above are well-balanced and representative of all labels. Nevertheless, it was necessary to preprocess the dataset through pipelines to fit the data into networks. The type of data fed to the model varied greatly according to its type and whether the algorithm of choice incorporated the temporal dynamics. *Torchvision* library were very helpful in creating preprocess pipelines that allowed the right data format and data augmentation.

Finally, the canonical split between the test set and the train set has been made to test the result. On the other hand, the evaluation set was not used to test the adult tolerance of the models it did not require the perfect tuning of the model.

4.1.3 Feeding data to networks

A critical part of the thesis was how to feed data to networks to guarantee similar accuracies between the same dataset across different models. In general, given classical models such as CNNs and FNNs, it is enough to feed images to the former

and flatten images for FNNs. The MNIST is a dataset of images, while SHD is made of arrays of inputs, which are converted into images before using them.

When dealing with SNNs a further step is needed since they implement temporal dynamics and need therefore temporal data. NMNIST entry is already divided into time windows, each of n time steps. It is then enough to set the forward function to pass a frame (out a whole video) at a time for both spiking FNN and spiking CNN. Of course, for FSNN a flattened image is passed at each time step of the time window (For CSNN the same is true but without the flattening operation). For SHD and FSNN it is simple enough as well: an array is passed at every time step. To adapt SHD dataset to CSNN it was needed to first group all the arrays in a single image representing an element of the dataset. But since this image is static (each image represents a whole time window), to mimic a temporal dynamic, it is fed to the network unchanged n times mimicking a static video.

4.1.4 Neural Networks

Each architecture's design was created considering the constraints derived from the simulation time. The more parameters are present in the network, the longer the fault injection campaign takes. Therefore, optimizing each NNs with a limited number of layers was necessary. This was especially true for CFNNs since convolutional layers have many parameters. For this reason, the maximum depth of any SNN designed is 4 core layers (meaning convolutional or fully connected layers). In addition, while building SNNs for N-MNIST and SHD datasets didn't require many optimizations, Classical NN needed few adjustments and extra layers, to reach comparable accuracies. In particular, Batch Normalization and Relu layers were added to stabilize learning and increase accuracy.

The loss function used for NNs is *CrossEntropyLoss*, combined with *Adam* optimizer. Instead, SNNs needed domain-specific loss functions such as *mse_count_loss*.

When called, the total spike count is accumulated over time for each neuron. The target spike count for correct classes is computed as the number of steps multiplied by the correct rate, while for incorrect classes it is the number of steps * incorrect rate. The spike counts and target spike counts are then applied to a mean square error loss function.

In appendix A each model structure on different datasets is highlighted. It can be noticed that not neuromorphic models are more complex even if they retain the same number of core layers (Convolutional and Linear ones) as their neuromorphic versions. In conclusion, it must be underlined that the optimization of the selected models does not aim at reaching State-of-the-art accuracy on the three benchmarks but at evaluating the impact of the fault injection through ANNFI.

4.1.5 Fault tolerance metrics

To properly evaluate the simulation result, a few metrics were chosen to reflect the effect of fault injections on the model prediction. A model output can be interpreted as probabilities, each corresponding to a label. They indicate how likely the network input corresponds to each class. A good system to evaluate whether the fault injected was very effective is to track how much the probabilities mentioned changed after the injection. In particular, it was chosen to consider only the variation of top-ranked probability between a golden run and a faulty one. In this context the Silent Data Corruption (SDC) term is key. It stands for errors in the data output that may go undetected unless the system in which the model is deployed systematically checks for them. In general, the variation of output probabilities is classified into the following types:

- **Masked**: The output top-ranked probability of the model remains unchanged after the fault injection. It is called masked because the network can mask the effect of the fault.

- **Non-critical SDC**: This kind of network corruption does alter the top-ranked output probability but not significantly enough to change the final prediction.
- **SDC-1**: Modifies the top-ranked output probability enough to change the classification result. These kinds of corruption are the most critical.

4.2 Experimental Results

In this section simulation times and simulations reliability are examined and commented on. It then follows a comprehensive analysis of the experimental results attempting to point out the main weakness of the models considered.

4.2.1 Fault injection times

Each simulation takes a few hours. This number may vary a lot according to the number of faults injected, the dimension of the network, and the data fed as input. In table 4.2.1 are shown the fault injection time:

Network Model	Dataset	Injected Faults	FI times [h:m:s]
CNN	SHD	16895	1:50:24
CSNN	SHD	7118	8:58:48
FNN	SHD	89957	4:59:43
FSNN	SHD	16876	2:04:05
CNN	MINST	16864	0:24:41
CSNN	NMINST	3022	1:03:36
FNN	MINST	16709	0:21:24
FSNN	NMINST	16840	2:00:42

Table 4.2: Models accuracies on datasets

In general spiking neural networks take more time to simulate. The only exception is the SHD dataset for FNN, where images as big as 250x250 pixels have been fed to the network consequently increasing simulation time. In addition, the

number of faults injected is very high since the number of injectable parameters is very high. Overall the simulation time is short enough to allow for extensive exploration of the fault space of many parameter networks.

4.2.2 Simulations reliability

It is important to underline that any simulation performed gives similar results if repeated multiple times. This is because a representative portion of each model parameter type has been fault-injected according to SFI formulas mentioned in 3.1.

The following tables show the injection details for each model. The percentage taken into account changes a lot depending on the numerosity of the parameter examined. For example, if there are 100000 *fc.weights* only a small percentage of them can be fault-injected otherwise the simulation would take too long. On the other hand, if the numerosity of *conv.bias* is minimal, all of them can be examined, even if not needed to reach a good representativeness of the set.

Table 4.3: Injection details for different models

Model	Name	n_injected	total	percentage_injected (%)
CNN_0_71_SHD	conv1.bias	30	32	93.75
	conv1.weight	25	128	19.53
	conv2.bias	31	32	96.88
	conv2.weight	1156	4096	28.22
	conv3.bias	29	32	90.63
	conv3.weight	1188	4096	29.00
	conv4.bias	16	16	100.00
	conv4.weight	608	2048	29.69
	fc.bias	20	20	100.00
	fc.weight	11056	38720	28.55
CNN_0_83_MNIST	conv1.bias	24	24	100.00
	conv1.weight	218	600	36.33
	conv2.bias	40	48	83.33
	conv2.weight	9403	28800	32.65
	fc.bias	10	10	100.00
	fc.weight	3984	12000	33.20
CSNN_0_66_SHD	conv1.bias	6	6	100.00
	conv1.weight	67	96	69.79
	conv2.bias	24	24	100.00
	conv2.weight	26	2304	1.13
	fc.bias	20	20	100.00
	fc.weight	2776	1060320	0.26
	lif1.beta	66	98	67.35
	lif1.mem	869	57624	1.51
	lif1.spk	899	57624	1.56
	lif1.threshold	55	98	56.12
	lif2.beta	42	47	89.36
	lif2.mem	805	53016	1.52
	lif2.spk	815	53016	1.54
	lif2.threshold	41	47	87.23
	lif3.beta	20	20	100.00
	lif3.mem	8	20	40.00
	lif3.spk	7	20	35.00
	lif3.threshold	20	20	100.00

Model	Name	n_injected	total	percentage_injected (%)
CSNN_88_NMNIST	conv1.bias	2	2	100.00
	conv1.weight	64	100	64.00
	conv2.bias	4	4	100.00
	conv2.weight	131	200	65.50
	fc.bias	10	10	100.00
	fc.weight	661	1000	66.10
	lif1.beta	27	28	96.43
	lif1.mem	247	392	63.01
	lif1.spk	270	392	68.88
	lif1.threshold	27	28	96.43
	lif2.beta	10	10	100.00
	lif2.mem	59	100	59.00
	lif2.spk	65	100	65.00
	lif2.threshold	10	10	100.00
	lif3.beta	10	10	100.00
	lif3.mem	6	10	60.00
	lif3.spk	4	10	40.00
	lif3.threshold	10	10	100.00
FNN_0_80_SHD	fc1.bias	195	4000	4.88
	fc1.weight	50630	40000000	0.13
	fc2.bias	54	1200	4.50
	fc2.weight	36038	4800000	0.75
	fc3.bias	20	20	100.00
	fc3.weight	991	24000	4.13
FNN_0_96_MNIST	fc1.bias	11	1024	1.07
	fc1.weight	9877	1048576	0.94
	fc2.bias	7	682	1.03
	fc2.weight	6592	698368	0.94
	fc3.bias	10	10	100.00
	fc3.weight	70	6820	1.03

Model	Name	n_injected	total	percentage_injected (%)
FSNN_0_82_SHD	fc1.bias	16	300	5.33
	fc1.weight	12254	210000	5.83
	fc2.bias	7	200	3.50
	fc2.weight	3509	60000	5.85
	fc3.bias	20	20	100.00
	fc3.weight	253	4000	6.33
	lif1.beta	22	300	7.33
	lif1.mem	16	300	5.33
	lif1.spk	16	300	5.33
	lif1.threshold	16	300	5.33
	lif2.beta	6	200	3.00
	lif2.mem	6	200	3.00
	lif2.spk	11	200	5.50
	lif2.threshold	11	200	5.50
	lif3.beta	20	20	100.00
	lif3.mem	6	20	30.00
	lif3.spk	7	20	35.00
	lif3.threshold	20	20	100.00
FSNN_0_95_NMNIST	fc1.bias	9	200	4.50
	fc1.weight	14472	204800	7.07
	fc2.bias	12	100	12.00
	fc2.weight	1393	20000	6.97
	fc3.bias	10	10	100.00
	fc3.weight	63	1000	6.30
	lif1.beta	15	200	7.50
	lif1.mem	13	200	6.50
	lif1.spk	12	200	6.00
	lif1.threshold	11	200	5.50
	lif2.beta	6	100	6.00
	lif2.mem	5	100	5.00
	lif2.spk	12	100	12.00
	lif2.threshold	4	100	4.00
	lif3.beta	10	10	100.00
	lif3.mem	6	10	60.00
	lif3.spk	4	10	40.00
	lif3.threshold	10	10	100.00

4.2.3 SDC1

Concerning the data obtained, the first element to notice is that in general SNNs are more robust to fault injection as shown in figure 4.1. The data gathered generally seems to agree with [23], where critical faults lie in the range $[0\% - 4\%]$ for each network examined.

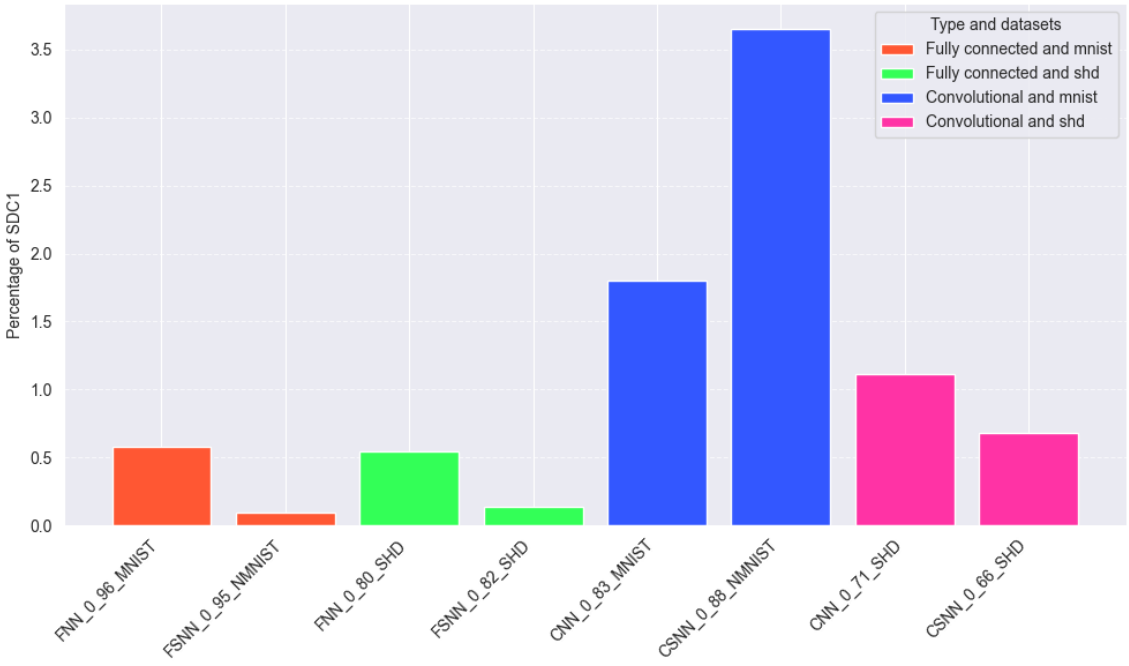


Figure 4.1: Comparison of SDC1 percentage for every network. Each color represents a type of network and dataset.

Three out of four examples show that spiking neural networks are more resilient to changes, in particular in the case of fully connected Neural networks which are 6x times more resilient. The only exception is the convolutional neural network deployed on mnist. The spiking version has a higher percentage of SDC1, meaning the resulting prediction changed 3.6% after the fault injection. It can also be noticed that CNN on mnist has the highest number of critical faults.

This makes particular sense with figure 4.2 where it is clearly shown that

convolutional and Leaky layers are the most influenced by faults. Therefore, networks built with the above-mentioned types of layers for the most part are shown to be most damage-prone in 4.1.

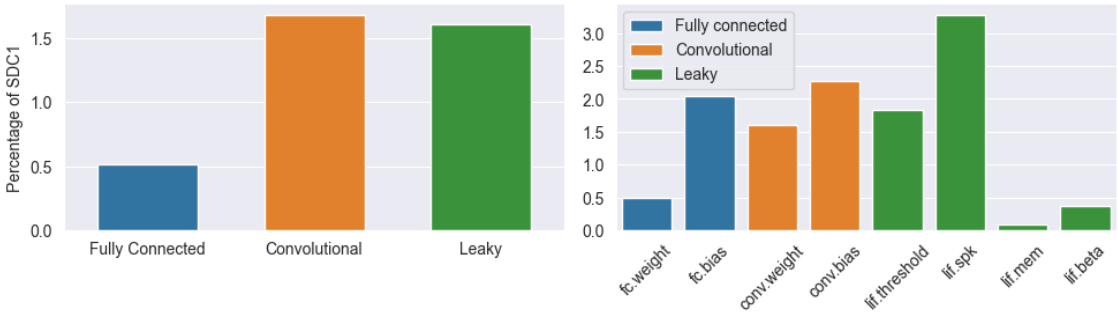


Figure 4.2: Comparison between SDC1 percentage caused by stuck-at-1 faults injections. Plots have a type of layer granularity on the left vs a type of parameter granularity on the right

Nonetheless, It is not clear why CSNN_0_88_NMNIST is less fault tolerant than its classical version. One possible explanation can be attributed to the difference in dimension between CSNN_0_88_NMNIST and CSNN_0_66_SHD since the former is much bigger. A small number of neurons may cause decreased resiliency: Injecting a fault in a large network may cause less damage since many other neurons contribute to the final prediction together with the injected one. Therefore, while CNN and CSNN on MNIST retain similar dimensions, CNN and CSNN on SHD have very different numbers of neurons which results in CSNN_0_66_SHD being more resilient.

4.2.4 Spiking neural network overall results

In 4.2 another important element is shown: **spikes, thresholds, and bias are the most critical parameters to inject.** This trend is further confirmed by Figure 4.3, 4.4, 4.5, 4.6 where all SNNs parameters are shown together with SDCs.

One possible explanation is that setting a spike to 0 or 1 during inference time

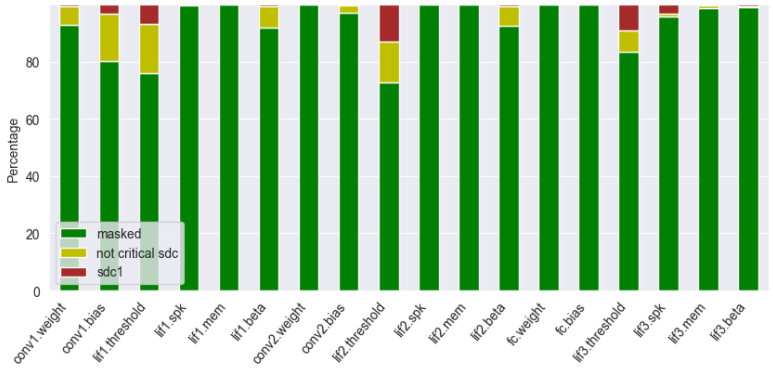


Figure 4.3: Convolutional spiking neural network overall classification results on SHD

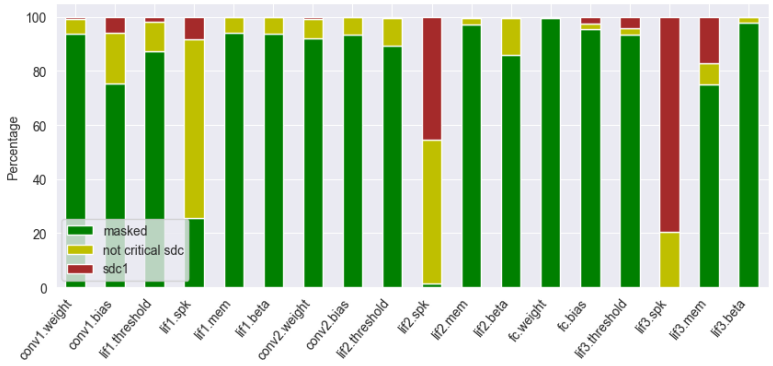


Figure 4.4: Convolutional spiking neural network overall classification results on NMNIST

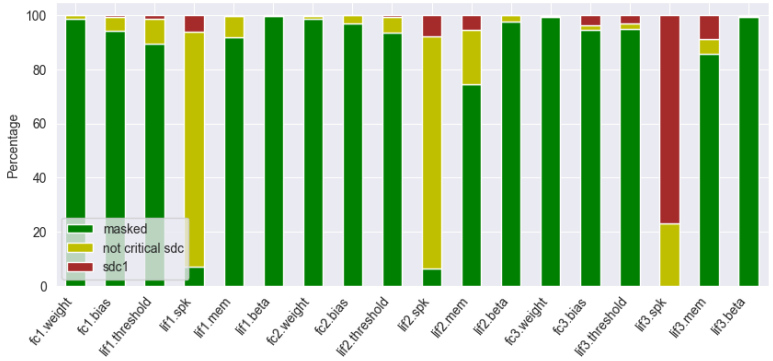


Figure 4.5: Feed Forward spiking neural network overall classification results on SHD

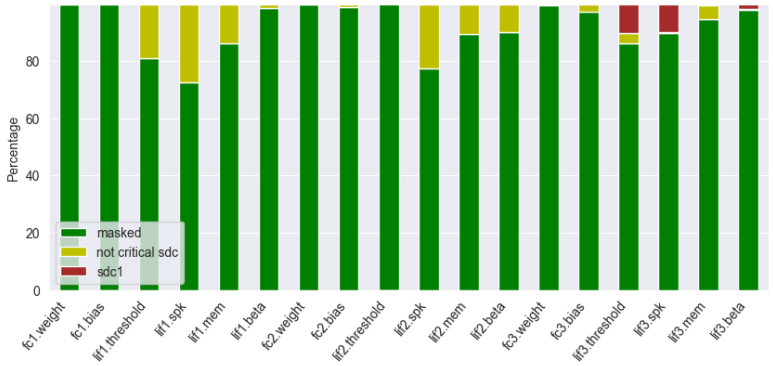


Figure 4.6: Feed Forward spiking neural network overall classification results on NMNIST

is equivalent to a powerful signal to the following layers that a feature is recognized in the image. This is particularly true in the network's last layer where the larger sum of the spikes relative to a class is the network's prediction. Also drastically reducing or increasing threshold parameters has the same result as injecting faults in spikes. This is because setting a low threshold is equivalent to always firing spikes and vice versa: the slightest signal would increase $U(t)$ enough to reach the threshold and consequently trigger the firing of a spike or never trigger it if the fault injected increased the threshold.

As already mentioned, fault-injecting bias values are impactful on network performances. This phenomenon can be explained by the relatively few biases in comparison with weight parameters, even in the same layer. This is exemplified by the following equation describing the output of a neuron:

$$f(b + \sum_{i=1}^{n} x_i w_i) \tag{4.1}$$

with x_i being the input, w_i the weights and b and f respectively bias and activation function. For any b there are n weights, therefore, injecting any of the biases results in a larger effect compared to the one obtained by injecting one of the many weights. A similar logic can be used to explain why **Fully connected layers are usually more resilient convolutional layers**. Convolutional weights are in fact usually less then fully connected weights. Therefore, modifying a convolutional weight results in more corruption caused to the network's performances as shown in Figure 4.2.

4.2.5 Neural network overall results

The plots below refer to classical neural network fault injection. They are very different from the SNNs results: The percentage of non-critical SDC increased greatly. This highlights once again that SNNs tend to be more robust, not just

against SDC1 faults. The tendency of NNs to produce a higher rate of non-critical SDCs can be attributed to the differences in the nature of their final outputs compared to SNNs. For SNNs the final output is the spike count while probabilities are the outputs of classical NNs. The former are integer numbers and the latter are floats with many significant figures. Therefore likely to have a non-critical SDC on NNs since even the slightest change in the probability is accounted for by the output float values and thereby classified as non-critical SDC or SDC1. Instead, It is more difficult to cause a non-critical SDC in SNNs because the effect of the fault on the output must be of at least a spike fired ($+1$) or not fired (-1) to be accounted for as non-critical SDC.

Analyzing further the results in Figure 4.7 and Figure 4.8 obtained for CNNs on MNIST and SHD we can observe a huge discrepancy between the fault resiliency of these two networks even if they belong to the same type of network. It can be explained by the difference in architecture as shown in the appendix A: A deep CNN had to be developed to manage the large images from SHD dataset. Instead, less than half the layers were enough to train a performing CNN on MNIST. We can thus conclude that more complex networks obtain better fault resilience overall. This is further corroborated by Figure 4.9 and Figure 4.10 where the larger FNN deployed on SHD seems to have a lower rate of non-critical faults even if the difference is less marked.

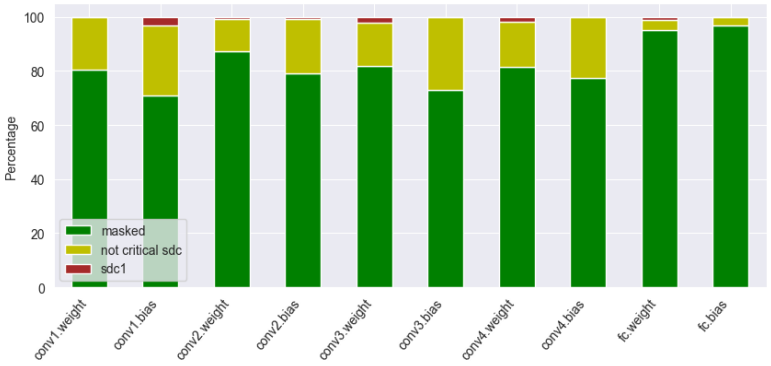


Figure 4.7: Convolutional neural network overall classification results on SHD

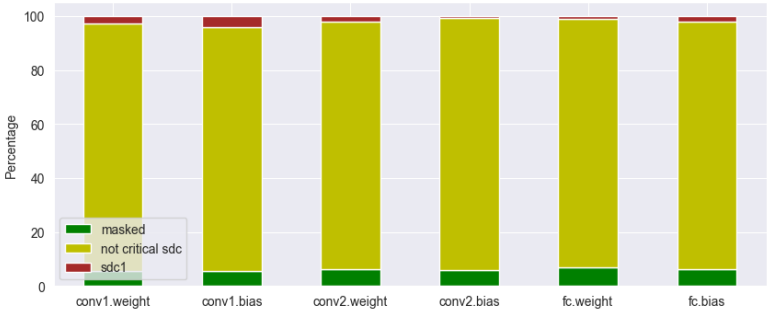


Figure 4.8: Convolutional neural network overall classification results on MNIST

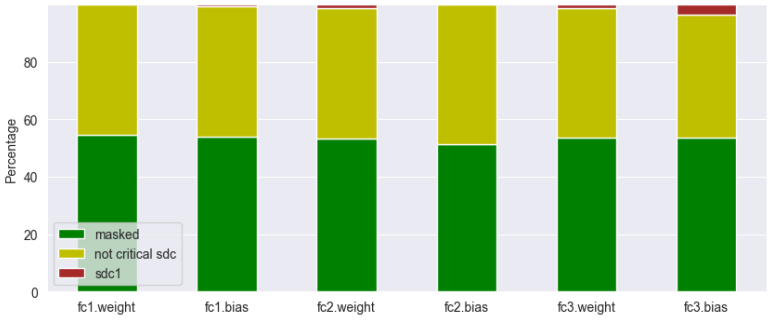


Figure 4.9: Feed Forward neural network overall classification results on SHD

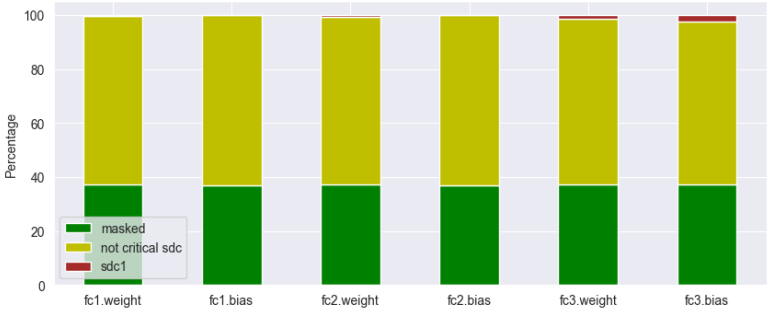


Figure 4.10: Feed Forward spiking neural network overall classification results on MNIST

4.3 Comparison between stuck-at-1 and stuck-at-0 fault

As it's possible to see from Figure 4.11 and 4.12 non-critical SDC and masked values are in the same proportions for both stuck-at-0 and stuck-at-1 faults injection. The only percentage changing is SDC1 values, which are considerably more present in stuck-at-1 fault injection scenario. This corroborates the scenario described above, where spiking neural networks generally performed better when fault-injected.

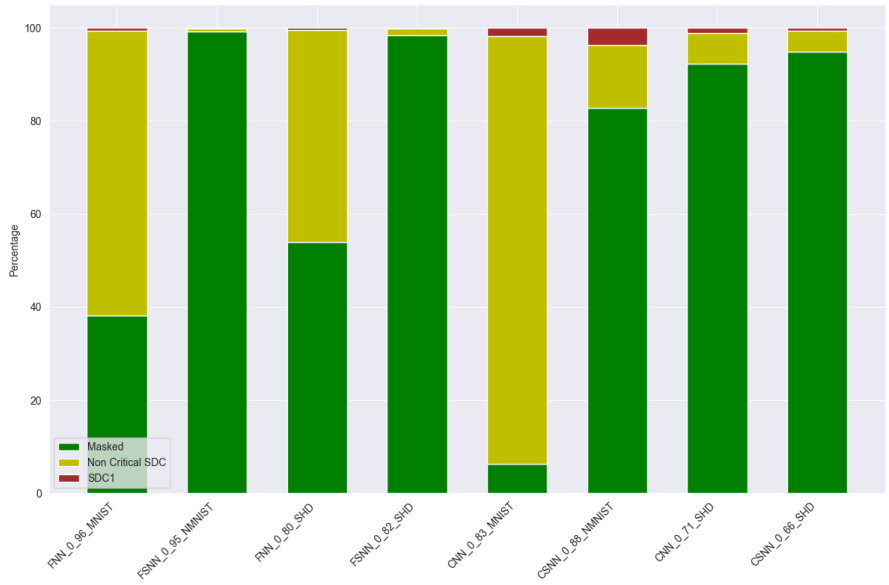


Figure 4.11: Overall classification results for every network after stuck-at-1 fault injection

Dead neurons and distribution of weights

Looking at the above graphs one question arises: **why do non-critical SDCs and Masked percentages remain very similar while SDC1 reduces significantly**? To understand this we have to look at more in-depth in the data. From SDC1s in Figure 4.13 we can see a large variation in the parameters and layers

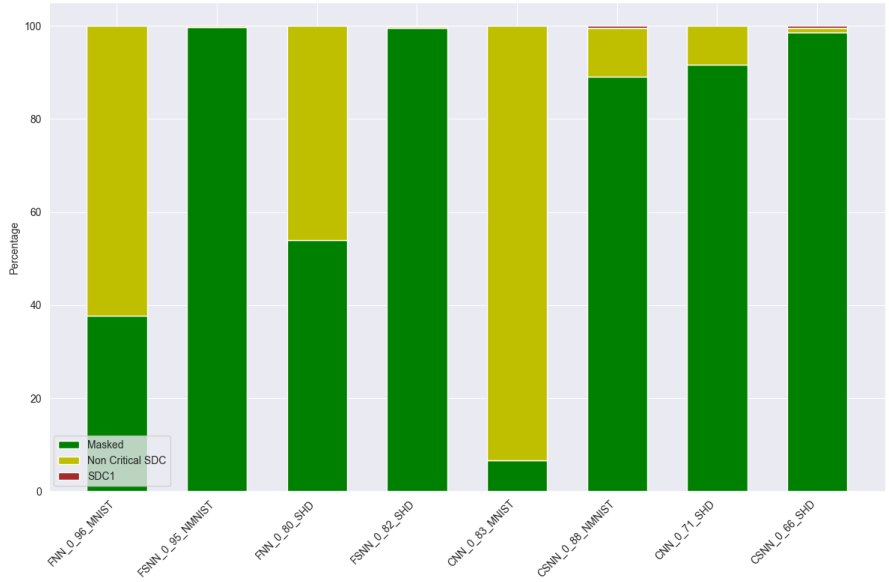


Figure 4.12: Overall classification results for every network after stuck-at-0 fault injection

responsible for SDC1s in comparison with 4.2. Fully connected and convolutional layers impact very little SDC1s percentage, while Leaky layers, especially threshold parameters, are nearly the only ones responsible for SDC1.

As already mentioned, a threshold is a tricky parameter, which, if it is set to a too large value, might metaphorically kill a neuron, inhibiting any spike activity.

That is the case of Stuck-at-1 fault injections. On the contrary, setting a threshold too low may cause an always-active neuron. In the stuck-at-0 scenario, we can hypothesize that many neurons are made hyperactive by reducing their thresholds, thereby explaining why threshold fault injections seem to be the main cause of SDC1. On the other hand, spikes are often zeros, and setting their value does not imply large SDC1 differences.

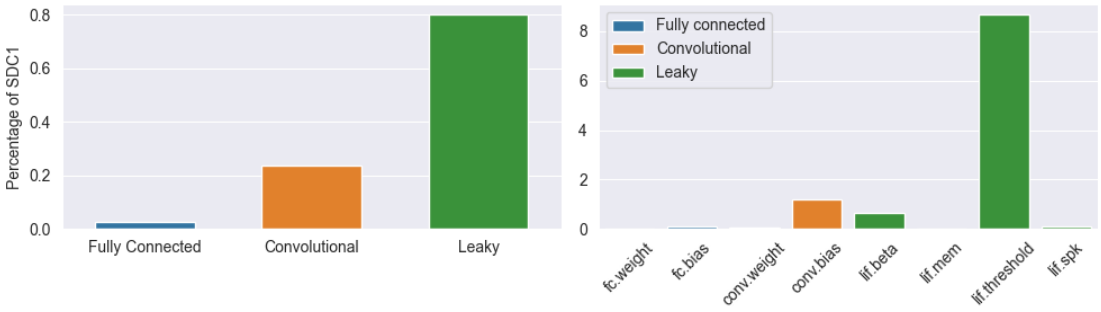


Figure 4.13: Comparison between SDC1 percentage caused by stuck-at-0 faults injections. Plots have a type of layer granularity on the left vs a type of parameter granularity on the right

Last but not least, **what is causing the near-to-zero effect of injecting faults in Fully connected and Convolutional layers?** To answer the question weight values distribution plots of the two types of layers must be taken into account. One possible example is the distribution of the network CNN_0_83_MNIST (figure 4.14).

It is possible to see a distribution with mean of 0.001. The same stands for all the other networks as shown in the appendix B. Given this, it can be explained why SDC1 values are mostly not caused by the above-mentioned layers: By injecting stuck-at-0 faults, weight floats are set to a value nearer to 0 compared to the golden one and since most of the values are by default near to 0, the overall impact is not significant. On the other hand, stuck-at-0 injections still cause an important percentage of non-critical SDCs. This might be because setting a bit to zero causes weight values far from the mean to change. Nonetheless, the process happens in a lesser degree since the standard deviation is not large and it causes for the most part only non-critical SDCs.

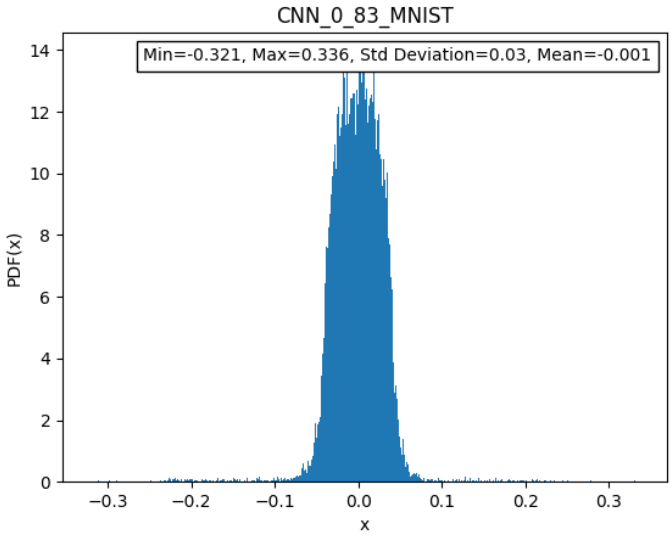


Figure 4.14: Weight value distribution of the convolutional neural network on mnist

Chapter 5

Conclusions

This thesis aims to analyze and comprehend the differences in resilience between spiking neural networks (SNNs) and classical neural networks (CNNs). This result was achieved through the modification of the ANNFI tool to support SNNs, as well as the implementation of various data analysis tools. As a result, the newly adapted ANNFI tool is now capable of assessing the resilience of fully connected and convolutional SNNs to faults, offering a platform that induces errors and injects faults into critical components such as synaptic weights, neuron model parameters, internal states, and activation functions.

Numerous simulations were performed using these tools, revealing that SNNs tend to demonstrate superior performance following fault injections, as indicated by various types of Silent Data Corruptions (SDCs). To sum up, a few key findings emerged during this research:

- **Biases, convolutional weights, thresholds, and spikes** were identified as the most critical parameters for fault injection.
- **Leaky layers were observed to enhance network resilience**, as they tend to mitigate the overall impact of stuck-at faults on the network, despite

their susceptibility to stuck-at-0 faults.

- Generally, **as the depth and size of the network increase, the impact of a fault is reduced**, as the significance of each neuron diminishes. It is essential to avoid layers with a small number of parameters since faults in these layers can disproportionately affect the network's final output.
- Reducing the number of convolutional layers improves the overall resiliency of the network.

It is important to note that **activation functions in classical neural network models were not subjected to fault injection**. In contrast, SNNs are examined from this point of view since Leaky layers already incorporate non-linearity through Leaky layers. Therefore SNNs resilience is further underscored.

5.1 Future Developments

These findings can serve as a foundation for future research, especially as real-world scenarios involve many more variables. Additional benchmark datasets could be utilized to validate these results, and comparisons with other models that incorporate temporal dynamics, such as Recurrent Neural Networks (RNNs), could provide further insights. Finally, extending fault injection to the activation functions of classical neural networks presents a promising direction for continuing this work.

Appendix A

Models Architecture

Spiking CNN on SHD		
Conv2d	6 filters	4x4 kernel size
MaxPool2d	2x2 pool size	//
Leaky	//	//
Conv2d	24 filters	4x4 kernel size
MaxPool2d	2x2 pool size	//
Leaky	//	//
Linear	53016 nodes	//
DropOut	//	//
Leaky	//	//

Spiking CNN on NMNIST		
Conv2d	2 filters	5x5 kernel size
Leaky	//	//
MaxPool2d	2x2 pool size	//
Conv2d	4 filters	5x5 kernel size
Leaky	//	//
MaxPool2d	2x2 pool size	//
Linear	100 nodes	//
Leaky	//	//

Spiking FNN on NMNIST	
Linear	200 nodes
Leaky	//
Linear	100 nodes
Leaky	//
Linear	10 nodes
Leaky	//

Spiking FNN on SHD	
Linear	300 nodes
Leaky	//
Linear	200 nodes
Leaky	//
Linear	20 nodes
Leaky	//

Table A.1: Spiking neural networks architectures

CNN on SHD		
Conv2d	32 filters	2x2 kernel size
Relu	//	//
MaxPool2d	2x2 pool size	//
Conv2d	32 filters	2x2 kernel size
Relu	//	//
BatchNorm2d	32 nodes	//
MaxPool2d	2x2 pool size	//
Conv2d	32 filters	2x2 kernel size
Relu	//	//
BatchNorm2d	32 nodes	//
MaxPool2d	2x2 pool size	//
Conv2d	16 filters	2x2 kernel size
Relu	//	//
BatchNorm2d	16 nodes	//
MaxPool2d	2x2 pool size	//
ReLu	//	//
Linear	20 nodes	//
Softmax	//	//

CNN on NMNIST		
Conv2d	24 filters	5x5 kernel size
MaxPool2d	2x2 pool size	//
Conv2d	48 filters	5x5 kernel size
MaxPool2d	2x2 pool size	//
ReLu	//	//
Linear	1200 nodes	//
Softmax	//	//

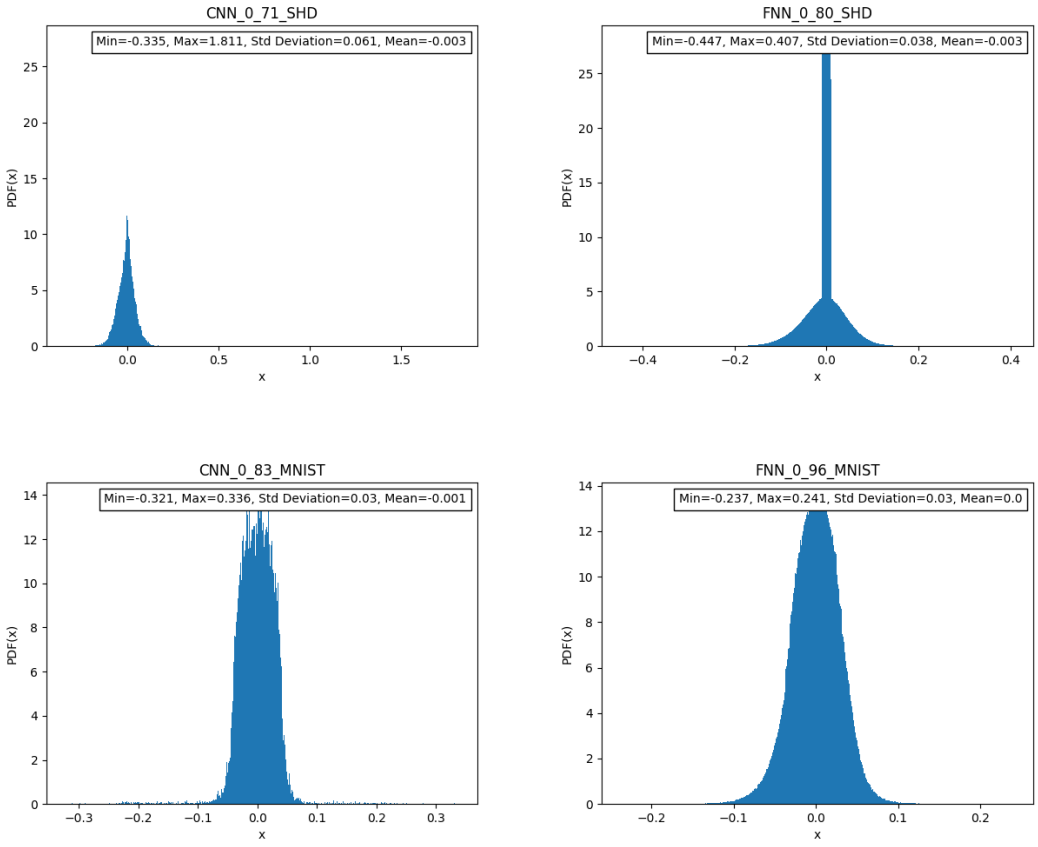
FNN on NMNIST	
Linear	1024 nodes
ReLu	//
Linear	682 nodes
ReLu	//
Linear	10 nodes
Softmax	//

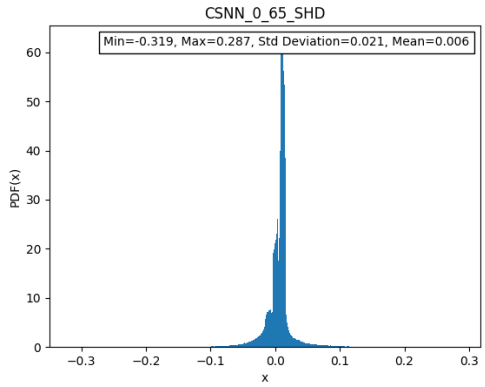
FNN on SHD	
Linear	4000 nodes
ReLu	//
BatchNorm1d	4000 nodes
Linear	1200 nodes
ReLu	//
BatchNorm1d	1200 nodes
Linear	20 nodes
Softmax	//

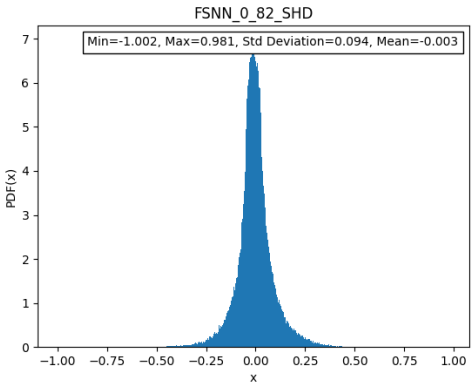
Table A.2: Classical Neural Networks architectures

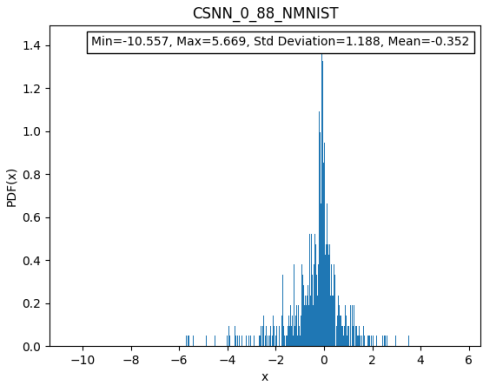
Appendix B

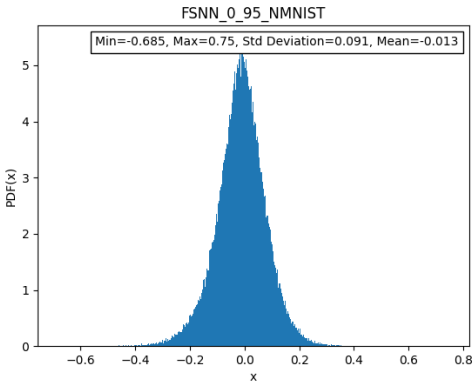
Weights distributions




CSNN_0_65_SHD
Min=-0.319, Max=0.287, Std Deviation=0.021, Mean=0.006
PDF(x)
x


FSNN_0_82_SHD
Min=-1.002, Max=0.981, Std Deviation=0.094, Mean=-0.003
PDF(x)
x


CSNN_0_88_NMNIST
Min=-10.557, Max=5.669, Std Deviation=1.188, Mean=-0.352
PDF(x)
x


FSNN_0_95_NMNIST
Min=-0.685, Max=0.75, Std Deviation=0.091, Mean=-0.013
PDF(x)
x

Bibliography

[1] Fukushima, Kunihiko, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", Biological Cybernetics, pp. 193–202, Springer-Verla, 1980.

[2] Carpegna, Alessio and Savino, Alessandro and Di Carlo, Stefano, "Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks". 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 14-19, July 2022.

[3] Carpegna, Alessio and Savino, Alessandro and Di Carlo, Stefano, "Spiker+: a framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge". `http://arxiv.org/abs/2401.01141`, January 2024.

[4] Padovano, Dario and Carpegna, Alessio and Savino, Alessandro and Di Carlo, Stefano "SpikeExplorer: Hardware-Oriented Design Space Exploration for Spiking Neural Networks on FPGA", Electronics 13, no. 9: 1744. `https://www.mdpi.com/2079-9292/13/9/1744`, January 2024.

[5] W. McCulloch, W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity", Mathematical Biophysics, pp. 115–133, 1943.

[6] Rosenblatt Frank. "The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain". Psychological Review, pp. 386–408, 1958.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” in Proceedings of the IEEE, pp. 2278–2324, 1998.

[8] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. "The computational limits of deep learning", 2020.

[9] Dario Amodei and Danny Hernandez. "AI and compute". Online: https://openai.com/blog/ai-and-compute/. 2019.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind, Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners", 2020.

[11] A. Dequino et al., "Compressed Latent Replays for Lightweight Continual Learning on Spiking Neural Networks," 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Knoxville, TN, USA, 2024, pp. 240-245, doi: 10.1109/ISVLSI61997.2024.00052.

[12] Payal Dhar. "The carbon impact of artificial intelligence". Nature Mach. Intell., 423–5, 2020.

[13] Lasse F Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: "Tracking and predicting the carbon footprint of training deep learning models", 2020.

[14] Nicolas Brunel and Mark CW Van Rossum. Lapicque’s 1907 "From frogs to integrate-and-fire. Biological Cybernetics", pp. 337–339, 2007.

[15] Alan L Hodgkin and Andrew F Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". The J. of Physiol., pp. 500–544, 1952.

[16] Eshraghian, Jason K and Ward, Max and Neftci, Emre O and Wang, Xinxin and Lenz, Gregor and Dwivedi, Girish and Bennamoun, Mohammed and Jeong, Doo Seok and Lu, Wei D, Training "spiking neural networks using lessons from

deep learning". Proceedings of the IEEE 2023.

[17] "T On the Future of Training Spiking Neural Networks", A Bendig, Katharina A Schuster, René A Stricker, Didier, Conference Proceedings, ICPRAM, pp, 466-473, 2023.

[18] J. V. Carreira, D. Costa, and S. J. G, "Fault Injection Spot-Checks Computer System Dependability", IEEE Spectrum, pp. 50–55, 1999.

[19] R. Leveugle et al. “Statistical fault injection: Quantified error and confidence”. 2009 Design, Automation & Test in Europe Conference & Exhibition, pp. 502–506, 2009.

[20] Benjamin Cramer et al. “The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks”. IEEE Transactions on Neural Networks and Learning Systems, pp. 2744–2757, July 2022.

[21] Y. Lecun et al. “Gradient-based learning applied to document recognition”. Proceedings of the IEEE 86.11, November 1998.

[22] Garrick Orchard et al. “Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades”. Frontiers in Neuroscience 9, November 2015.

[23] Göğebakan, Anıl Bayram, et al. Spikingjet: "Enhancing fault injection for fully and convolutional spiking neural networks". IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS). IEEE, 2024. p. 1-7.

[24] snnTorch tutorial website, Jason K. Eshraghian, 2024, `https://snntorch.readthedocs.io/en/latest/index.html`