Computer Engineering Lab
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ee.et.tudelft.nl/

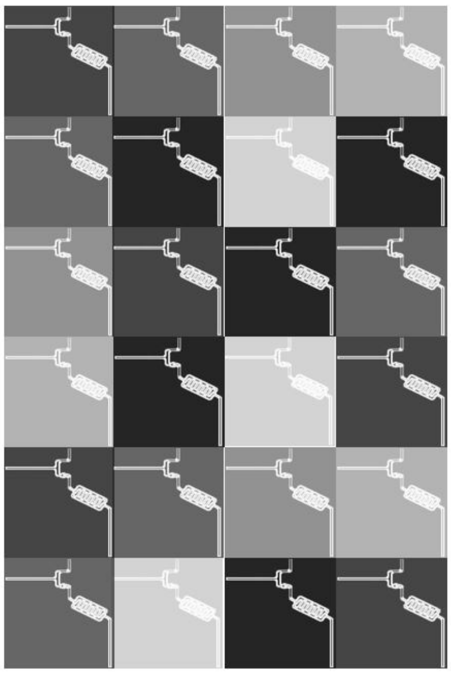
2021

MSc thesis

RRAM-based fault-tolerant Binary Neural Networks

Artemis Zografou

Abstract



Computation-In-Memory (CIM) employing Resistive-RAM (RRAM)-based crossbar arrays is a promising solution to implement Neural Networks (NNs) on hardware, such that they are efficient with respect to consumption of energy, memory, computational resources and computation time. In this respect, Binary NNs (BNNs), where the weights obtain single binary values, are inherently suitable for cost-effective CIM-based NN implementations. However, RRAM devices, due to variability and reliability issues, restrict the applicability of CIM-based NN. To address this issue and towards a low-cost NN hardware realization, in this thesis we: a) thoroughly investigate the impact of RRAM faults on the inference accuracy of RRAM-based BNNs, and b) propose three complementary fault-tolerance techniques to mitigate the impact of RRAM faults on the BNN's accuracy. These techniques are namely: a) a fault-tolerant activation function, b) a redundancy and weight range adjustment scheme, and c) a retraining technique. Evaluation results compiled on the MNIST, Fashion-MNIST and CIFAR-10 datasets demonstrate that the proposed techniques can improve the inference accuracy in the presence of RRAM faults by up to 20%, 40% and 80%, respectively. Moreover, comparisons with certain related state-of-the-art fault-tolerance frameworks indicate that the proposed techniques yield competitive results.

Faculty of Electrical Engineering, Mathematics and Computer Science

RRAM-based fault-tolerant Binary Neural Networks

by

Artemis Zografou
Embedded Systems: Embedded Computer Architectures
Student Number : 4951662
Email : a.zografou@student.tudelft.nl

Supervisor : Professor Said Hamdioui
Duration : June 2020 - July 2021

Committee members:
Said Hamdioui, CE, TU Delft
Rajendra Bishnoi, CE, TU Delft
Rene Van Leuken, CAS, TU Delft
Anteneh Gebregiorgis, CE, TU Delft

This work was performed in:

Computer Engineering Lab
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

To my parents Olga and Agis, and my sister Daphne.

Acknowledgements

I am excited to present you my master thesis on 'RRAM-based fault-tolerant Binary Neural Networks'. During my master thesis I had the chance to delve into the two areas I am most passionate about: emerging hardware architectures and neural networks. However, this rewarding journey, lasting around 1 year along with the Covid-19 pandemic, would not be possible without the support of the following people.

To begin with, I would like to thank my supervisor Prof. Said Hamdioui, who gave me valuable information regarding the thesis and helped me think out-of-the-box, by challenging me with interesting questions, when it mattered.

Moreover, I would like to thank my supervisor Prof. Rajendra Bishnoi, who gave me valuable advice and motivation throughout this thesis project. His contributions and help in setting the right goals for this thesis are appreciated.

I would like to deeply thank my daily mentor Anteneh Gebregiorgis for his invaluable support and help throughout this thesis project. Our regular discussions provided me with the guidance as well as a clear head, that I needed throughout stressful times in my thesis.

I would also like to thank Prof. Rene Van Leuken for accepting the invitation to be part of the thesis defence committee.

Finally, I would like to thank my family and friends for supporting and tolerating me throughout this thesis. Thank you for being there for me, listening to my concerns and helping me manage my anxiety.

Artemis Zografou
Delft, The Netherlands
August 30, 2021

Contents

List of Figures

1 Introduction

1.1 Motivation

Deep Neural Networks (DNNs) have shown phenomenal success in different application domains such as speech translation, object recognition and detection, computer vision and autonomous systems [12], [13]. In spite of their great performance characteristics, the computation and resource intensive nature of DNNs, such as high storage and bandwidth requirements, processor-centric computation and energy-consumption, hinder their deployment on hardware [14], [15].

Moreover, the increasing demand to deploy DNNs on resource constrained platforms such as edge devices (figure 1.1), further aggravates the challenge [16], [17]. These challenges, mainly the data and power bottleneck problems, become even more apparent when DNNs are implemented on von-Neumman computer architectures (the most widely used computer architectures). According to these architectures, systems are processor-centric, meaning that the processor is the main computing element and responsible for controlling all operations. In order for an operation to be executed the processor first requests all the needed data from the memory unit. This constant transaction between the processor and the memory results in 3 architectural challenges [18]:

- the *memory wall*: As shown in figure 1.2, there is an increasing distance in terms of speed between the central processor unit (CPU) and the memory. Every time a memory instruction is issued, the processor is waiting in an idle state for the information to be fetched from the memory unit. This gap, combined with the the need of DNNs' large amount of data, makes memory operations the most contributing factor in terms of latency.

- the *power wall*: Apart from increasing latency, the practical power limit for cooling is reached. This means that it is difficult to increase the CPU clock speed further. This trend can be seen in figure 1.3 where the CPU speed is gradually plateauing after the year of 2005.

- the *Instruction Level parallelism (ILP) wall*: Parallel execution using multi-thread and multi-core systems have increased the processing throughput of computing systems. However, figure 1.3 reveals that there is an increasing difficulty in finding enough parallelism in software programs, resulting in a performance slowdown.

Since matrix multiplication, the core operation of DNN applications, is the major contributor for the energy consumption and memory overhead of DNNs, Binary Neural Networks (BNNs) have been proposed to simplify and efficiently reduce the hardware requirements of DNNs, while achieving comparable inference accuracy [19]. BNNs use binary representation of neural weights (i.e., ± 1) instead of costly floating point weights,

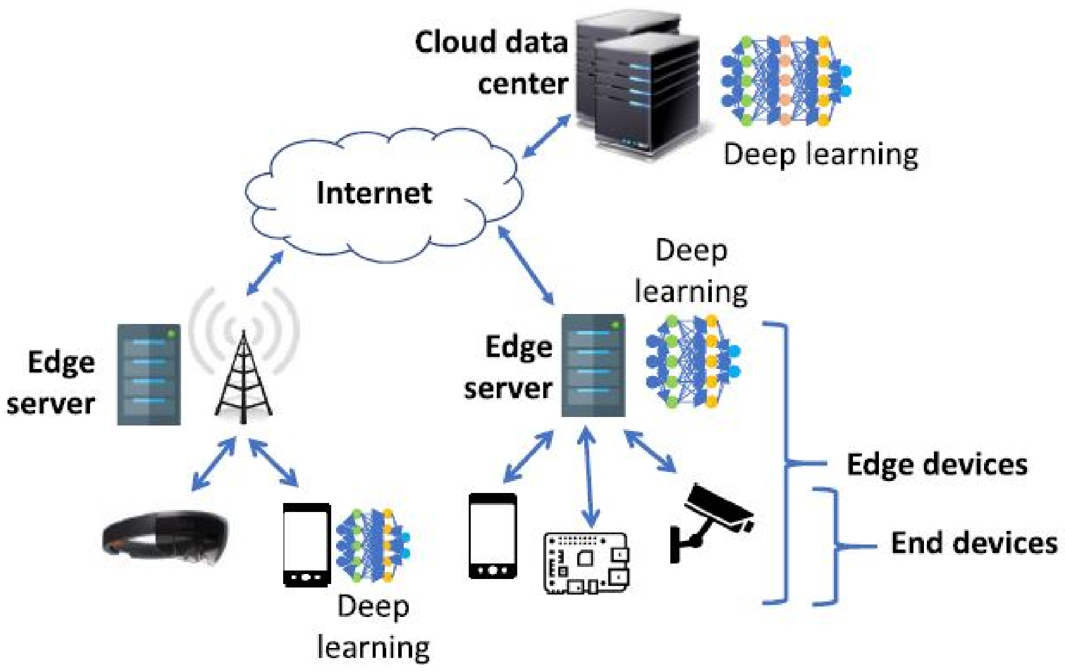


Figure 1.1: Deep learning can execute on edge devices and on cloud data centers [1].

in order to simplify the network's operations and peripheral circuits [19, 20]. Nonetheless, even though BNNs simplify the costly floating point matrix-vector multiplication, they still face the same challenges, when implemented on von-Neumman computer architectures and used for larger and complex network architectures.

To address the above, researchers have turned to Computation-In-Memory (CIM) architectures [14, 15, 21, 22, 23], in order to develop high-performance and energy-efficient hardware DNN accelerators (other approaches, with which this thesis is not concerned, include domain-specific FPGAs, CMOS, and non-CMOS based ASICs, etc. [24, 25, 26]). In CIM, memory acts both as a storage unit and as a computing element. Due to its inherent ability to perform computation on the stored data, CIM accelerates the Multiply-Accumulate (MAC) operation of DNNs (in fact, matrix-vector multiplication can be performed with $O(1)$ complexity) and circumvents the costly data movement of von-Neumann based systems [21]. Thus, especially when DNNs are considered, CIM reduces latency (*memory wall*) and power usage (*power wall*), and increases parallelizability (*ILP wall*). Consequently, CIM facilitates the realization of DNNs on various resource constrained platforms.

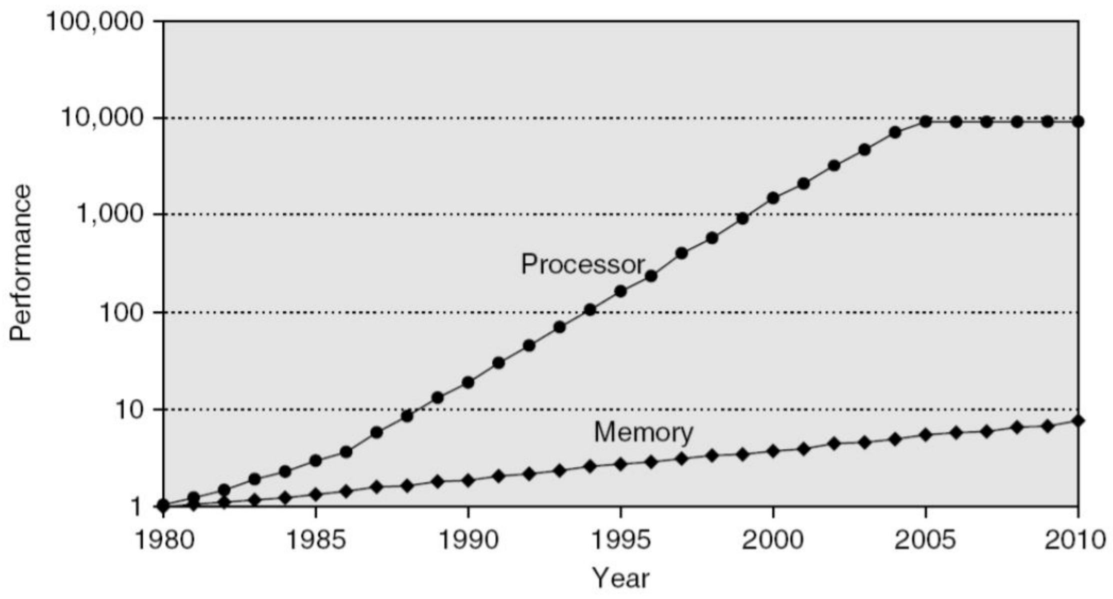


Figure 1.2: The memory wall [2].

1.2 Problem statement

An emerging memory technology enabling CIM is Resistive RAMs (RRAMs) [7]. In contrast to traditional memories such as DRAMs and SRAMs, in RRAM devices the information is stored as a high or low resistance value (HRS/LRS). RRAMs possess the following great advantages: they are non-volatile, more power-efficient and provide higher density integration. However, RRAMs have various reliability and variability problems that need to be overcome in order to be widely used. RRAM reliability issues can be classified into two main categories: 1) faults induced by the RRAM device's non-idealities, and 2) faults derived from the structure of the RRAM devices in crossbar arrays. The most common RRAM devices' reliability failures include conductance drift, read disturb, device variation etc [27, 28, 29, 30]. Permanent defects in RRAM devices, arising for example from the manufacturing stage, can be modeled by the Stuck-at-fault (SAF) model. In this case the device is stuck at a permanent low or high resistance state.

When DNNs are implemented using RRAM devices, the aforementioned faults degrade significantly the classification accuracy [11]. Hence, it is of critical importance to investigate fault tolerance techniques, for enabling the deployment of DNNs on CIM RRAM-based crossbar arrays. Several software and hardware-based fault tolerance approaches have been proposed [8, 31, 32, 9, 33]. However, these solutions have various limitations, such as significant software or hardware overhead, and need for complex mapping algorithms. Therefore, there is a clear need for a cost-effective and efficient fault tolerance framework in order to deploy DNNs on RRAM-based CIM crossbar arrays.

In addition, even though, as mentioned earlier, BNNs constitute a low-cost framework

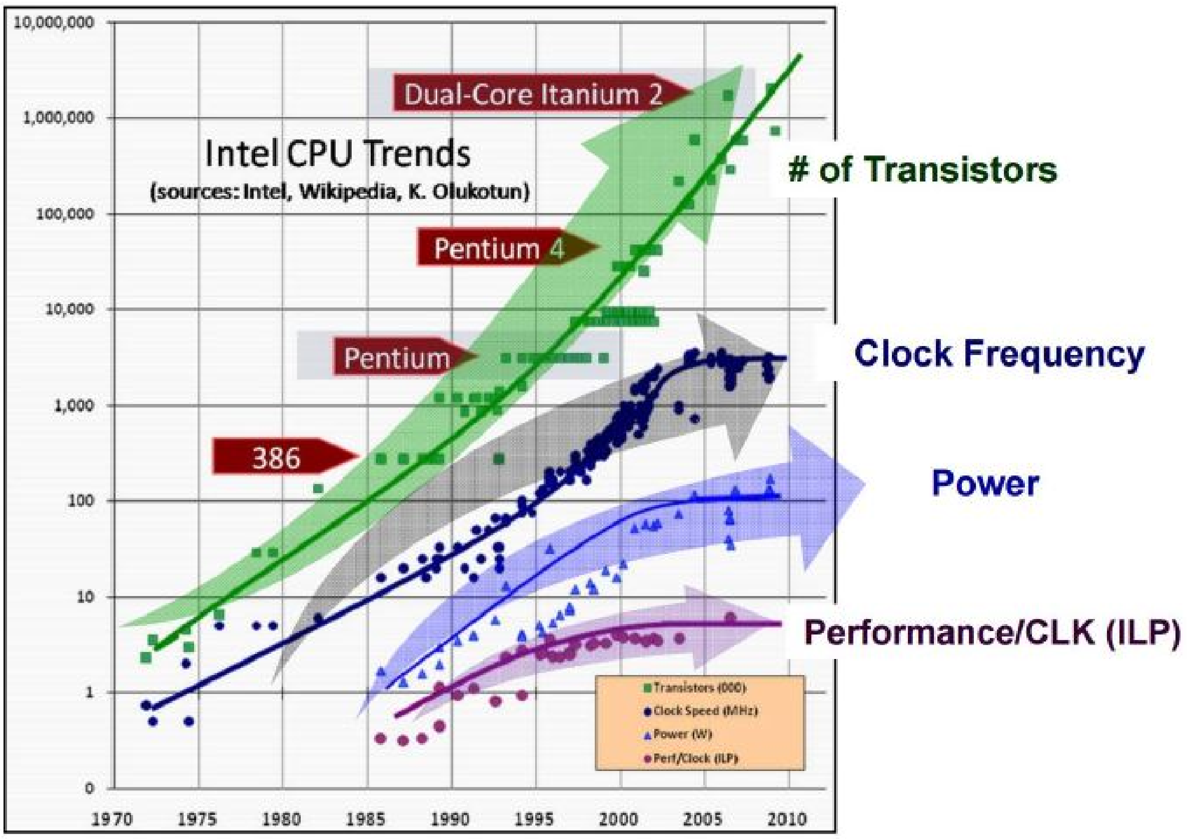


Figure 1.3: The power and ILP wall [3].

that reduces severely the hardware requirements of DNN hardware implementations, the literature on fault-tolerance techniques for BNN implementations on RRAM-based crossbar arrays remains scarce. In particular, the works in [34, 35, 36, 37] discuss implementation details of RRAM-based BNNs and do not propose any fault-tolerance techniques, while the fault-tolerance technique proposed in [38] suffer from certain limitations, which we thoroughly explain in Chapter 3.

The above give rise to the problem statement of the present thesis: *investigate the impact of RRAM-faults on the inference accuracy on BNNs implemented on RRAM-based crossbar arrays, and propose corresponding low-cost and effective fault-tolerance techniques.*

1.3 Contributions

In this work we propose cost-effective and efficient fault tolerance techniques addressing the impact of RRAM defect-induced Stuck-at-fault (SAF) and conductance variation on the inference accuracy of Binary Neural Networks (BNNs). First, the necessity for fault tolerance techniques on BNNs mapped to CIM crossbar arrays is motivated, by investigating the impact of RRAM faults and conductance variation on the inference accuracy of BNNs. Then, three fault tolerance techniques are proposed to mitigate the impact of RRAM faults and conductance variation on the BNN's inference accuracy. The first mitigation technique investigates the role of different activation functions on suppressing the impact of RRAM faults on the BNN's accuracy, and demonstrates how a fault-tolerance aware activation function can help to restore the accuracy. The second technique explores the potential of redundancy combined with a weight range adjustment for a fault-tolerant BNN design. Finally, the third technique constitutes a retraining method, to restore the BNN's accuracy.

The main contributions of this work are divided into two main parts:

1. The RRAM fault impact analysis demonstrates to what extent the accuracy of BNNs is influenced by the presence of RRAM faults (SAFs and conductance variation).

    - A low-cost architectural implementation using binary RRAM devices and BNNs is used. Compared to similar works [33, 8], where higher weight precision is assumed, the proposed architectural implementation of the RRAM based crossbar arrays utilizes less RRAM devices, as each weight is stored in 1 RRAM device.
    - A relatively large RRAM fault impact evaluation is conducted. In contrast to other works [32, 8, 33], where one dataset is considered, in this work, three different datasets, namely the MNIST, Fashion-MNIST and CIFAR-10 are used. Moreover, we evaluate the sensitivity of different BNN architectures architectures in the presence of RRAM faults.
    - We analyse the impact of SAFs and conductance variation on the inference accuracy of BNNs exclusively, as well as in a combined manner, for a variety of fault rates.

2. A fault tolerance framework is developed consisting of three complementary low-cost fault tolerance techniques are developed to mitigate the impact of RRAM faults on the BNN's inference accuracy, namely: a fault tolerant activation function, a redundancy and weight and adjustment scheme and a retraining method.

    - The fault tolerant activation function demonstrates the role of activation functions from a fault tolerance point of view. In particular, switching the activation function to *ReLU* improves the inference accuracy of the BNN by 10% and 15% for the MNIST [39], Fashion-MNIST [40] dataset respectively.
    - In redundancy and weight range adjustment, we utilize redundant devices not for remapping faulty devices -as previous works [31], [41] - but for extending the set of values that each weight can hold. As a result the inference accuracy is increased by 5% and 8% for the MNIST [39], Fashion-MNIST [40] dataset correspondingly.
    - Although similar retraining techniques have been proposed by the related literature [32], [8], they fail to restore completely the inference accuracy back

to the ideal value, when high number of faults occur. In this work, retraining recovers the inference accuracy in all 3 datasets very close to their fault-free accuracy value.

- Evaluation results show that the proposed techniques are able to achieve on average up to 20%, 40% and 80% inference accuracy improvement of a BNN mapped to a faulty RRAM crossbar array using the MNIST [39], Fashion-MNIST [40] and CIFAR-10 [42] datasets, respectively, while imposing negligible overhead.

1.4 Outline

The rest of the thesis is organized as follows. In chapter 2 we start by introducing basic background information that is needed in order to follow the story-line of this work. Essential information regarding Neural Networks architectures is presented, including the working principle, different Neural Networks' (NNs) architectures as well as applications and challenges. Emphasis to Binary Neural Networks (BNNs) is given, which are the target of this work. Afterwards, we discuss Computation-In-Memory (CIM) architectures, used for realizing NNs on hardware, highlighting the motivation around them, their basic operating principles, and the computational potential of realizing BNNs on CIM architectures. We conclude this chapter by introducing RRAM devices. The applications and benefits of RRAM devices are demonstrated, along with their reliability and variability problems.

Chapter 3 provides a thorough literature survey regarding the state-of-the-art related fault tolerance works. The fault tolerance techniques are divided into two categories: software and hardware based. Subsequently, advantages and limitations of each category are specified in this chapter.

The RRAM fault impact analysis is presented in Chapter 4. First, the evaluation setup is disussed, which is used for demonstrating the impact of RRAM faults on the inference accuracy of BNNs. The evaluation setup includes two main parts: the architectural implementation (how BNNs are implemented on RRAM crossbar arrays) and the simulation framework (how we simulate BNNs running on RRAM crossbar arrays and the corresponding RRAM faults). Finally, the fault impact evaluation results are presented.

Motivated by the fault impact analysis, our fault tolerance framework is introduced in chapter 5. We begin by discussing the role of activation functions from a fault tolerance point of view. Afterwards, we present a redundancy and weight range adjustment scheme. Finally, a modified retraining method is presented.

The simulation results of the fault tolerance framework are showcased in Chapter 6. The developed fault tolerant techniques are evaluated using the simulation framework build in Chapter 4. A discussion regarding the benefits and drawbacks of the proposed framework is presented, followed by a comparison with state-of-the-art works.

Finally, the work is concluded in Chapter 7, by providing an overview of the contributions and setting the ground for further improvements and future work.

Background 2

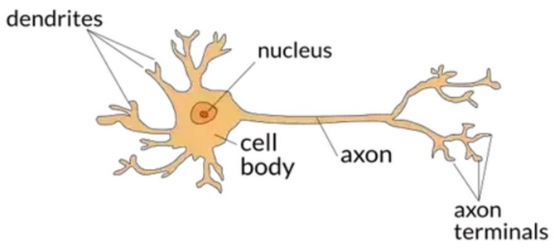
In this section the background information of neural networks, CIM and RRAM basics is provided. The chapter starts with introducing basic information as well as the applications and challenges of neural networks. After, the basics and applications of Computation-In-Memory (CIM) based NNs is discussed. Finally, we provide essential knowledge regarding RRAM devices, among others the working principle, fundamental defects, assets and challenges of using them.

2.1 Neural Networks (NNs)

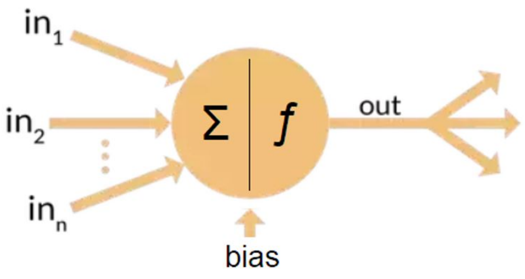
Neural networks are computing systems inspired by the way the human brain works. In general, a neural network consists of a number of nodes and edges; nodes represent the artificial neurons, whereas edges represent the way different neurons are connected to each other (weights). As displayed in Figure 2.1a, a biological neuron consists of the dendrites, the cell body, the axon and the axon's terminals [43]. Dendrites are responsible for transferring the received information from other neurons to the cell body. The cell body receives the information and forwards it through the axon. The axon normally ends with a number of synapses connecting to the dendrites of other neurons [44]. In the analogy of artificial and biological neurons, the dendrites can be considered as the input of the artificial neuron, the cell body the node, the axon the output of the artificial neuron and lastly the synapses can be regarded as the interconnections between the neurons. Similar to the human brain [45], an artificial neuron is the main component of a NN, while each connection/edge, representing a brain's synapses, transmits information from one neuron to another (Figure 2.1b). Each artificial neuron takes a number of inputs and process them by taking the weighted sum of the inputs and the corresponding weights. After that, a bias term is added to the weighted sum and then passed through a non-linear activation function. The produced output is fed into the next layer of connected as an input [46].

2.1.1 Basics

NNs or multi-layer perceptrons, are dense networks meaning that all neurons are fully connected and can consist of multiple cascaded layers. The number of layers as well as the number of neurons are optimized based on the corresponding application. A vanilla class of feed-forward NNs is the Multi-Layer Perceptron (MLP). A simple example of a 2-layer MLP can be seen in Figure 2.2. The NN consists of 3 layers: input, hidden and output layer. Except for the input nodes, each node is a neuron followed by a nonlinear activation function. Its multiple layers and non-linear activation make it capable of distinguishing data that are not linearly separable. Each edge has a weight W_{ji}, representing that the



(a) A biological neuron.



(b) An artificial neuron.

Figure 2.1: [4]

j-neuron is connected with the i-neuron through the weight W_{ji}. The output Z_j (j = 0, ...,m) of the hidden layer, is equal to the weighted sum of the input V_i (i=0,...,n) and the corresponding weights W_{ji} (j=0,...,m). Before applying the activation function f, the bias term b_j is added, as it can be seen in equation 2.1. Similarly, Y_i (i=0,...,k) is the output produced by the neurons in the output layer.

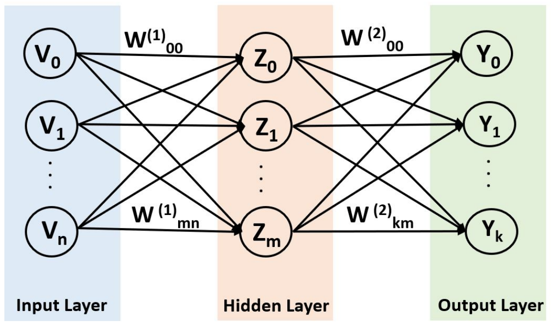


Figure 2.2: A 2-layer multi-layer perceptron structure.

$$Z_j = f(\sum_{i=0}^{n} W_{ji} \cdot V_i + b_j) \tag{2.1}$$

$$Y_j = f(\sum_{i=0}^{k} W_{ji} \cdot Z_i + b_j) \tag{2.2}$$

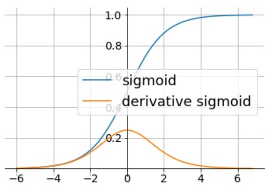
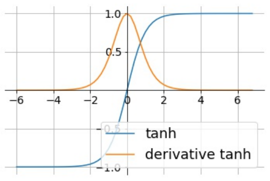
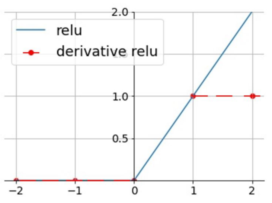
Activation function	**Description**	**Remarks**	**Graph**
Sigmoid	$s(x) = \frac{1}{1+e^{-x}}$	• bounds: [0, 1] • hidden layer • smooth gradient • non-zero centered • vanishing gradient • computational expensive	
Tanh	$t(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	• bounds: [-1, 1] • hidden layer • smooth gradient • zero centered • vanishing gradient • computational expensive	
ReLU	$r(x) = x^+ = max(0, x)$	• bounds: [0, +inf) • hidden layer • sparse • dying relu problem • computational efficient	
Softmax	$s(z)_i = \frac{e^{z_i}}{\sum_{j=0}^{k} e^{z_j}}$	• bounds: [0, 1] • output layer • multi-class problems	-

Table 2.1: Frequently used activation functions.

Activation functions are necessary, in order to add non-linearity to the network. This non-linearity helps the NN to be trained in complex non-linear input data. Broadly used activation functions are [47]:

- Sigmoid
- Hyperbolic Tangent (Tanh)
- Rectifier Linear Unit (ReLU)
- Softmax

Each activation function has its own features as well as exhibits different advantages and disadvantages. Some of them are presented in Table 2.1. The first 3 functions are mostly used for the hidden layers of the network, while softmax [46, Chapter 6.2.2.3] is used for the output layer. The Sigmoid and Tanh functions share a lot of the advantages/disadvantages, as Tanh is in fact a scaled Sigmoid function. However, Tanh is a zero-centered function, and has a stronger gradient than the Sigmoid, a feature that helps with the training of the network. The ReLU function is the most computationally efficient one, as simple mathematical operations are used (i.e. thresholding) compared to the other activation functions. Another benefit is that ReLU adds sparsity to the network, as some neurons might not be activated at all [48]. Therefore, ReLU makes the training easier as well as the network lighter. Nonetheless, a major problem with ReLU is the dying ReLU problem, which can be tackled by using a modified function, namely Leaky ReLU [49]. Contrarily, softmax is mainly used in the output layer of the network for multi-classification problems. The output of this function is a probability distribution along all the corresponding categories. The outputs take values in the range of [0, 1], with the sum of the probabilities being equal to 1. The target class of the softmax function scores the higher probability.

2.1.1.1 Training

In a nutshell, the goal of a neural network is to approximate some function f. As stated in [46], a feed-forward network defines a mapping y=f(x;θ) and learns the value of the parameters θ that result in the best function approximation. NNs are proven to be universal function approximators, meaning that they are capable of calculating any other computable function [50].

Therefore, training is an essential process for NNs, in order to be used to perform some meaningful tasks such as object detection and classification. There are two types of training: supervised and unsupervised. Briefly, unsupervised learning clusters unlabeled datasets, by discovering hidden patterns between the given data [51]. A common algorithm for unsupervised learning is clustering. In this thesis, supervised training is mostly discussed as unsupervised training in not relevant for this work. In supervised training, we train the NN by presenting several examples of a chosen dataset as well as the corresponding class that they belong to. In general, the objective of training is to tune the parameters of the network (weights and biases), in order to achieve high classification accuracy on a given dataset. The main algorithm used is back-propagation (BP) using gradient descent [46]. Several modifications and improvements regarding BP have also been proposed, such as Stochastic Gradient Descent (SGD) [52], Adam [53] etc. In BP, the goal is to adjust the weights (W_{ij}) and biases (b_{ij}), so that the cost/error function (E) is minimized in each step. The cost function shows how far the calculated output of the NN is from the target output, with respect to the corresponding network's parameters. In order to change the cost function with respect to the network's parameters, the chain rule is used, as it can be seen in equations 2.3, 2.4, 2.5, 2.6.

$$\frac{\partial E}{\partial W_{ij}^{(1)}} = \sum_k \left(\frac{\partial E}{\partial Y_k} \cdot \frac{\partial Y_k}{\partial Z_j}\right) \cdot \frac{\partial Z_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial W_{ij}^{(1)}} \tag{2.3}$$

$$\frac{\partial E}{\partial b_{ij}^{(1)}} = \sum_k \left(\frac{\partial E}{\partial Y_k} \cdot \frac{\partial Y_k}{\partial Z_j}\right) \cdot \frac{\partial Z_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial b_{ij}^{(1)}} \tag{2.4}$$

$$\frac{\partial E}{\partial W_{ij}^{(2)}} = \frac{\partial E}{\partial Y_k} \cdot \frac{\partial Y_k}{\partial W_{ij}^{(2)}} \tag{2.5}$$

$$\frac{\partial E}{\partial b_{ij}^{(2)}} = \frac{\partial E}{\partial Y_k} \cdot \frac{\partial Y_k}{\partial b_{ij}^{(2)}} \tag{2.6}$$

$$W_{ij(t+1)} = W_{ij(t)} - \eta \cdot \frac{\partial E}{\partial W_{ij}} \tag{2.7}$$

$$b_{ij(t+1)} = b_{ij(t)} - \eta \cdot \frac{\partial E}{\partial b_{ij}} \tag{2.8}$$

Equations 2.7, 2.8 show how the weights and biases are updated in each step of the gradient descent, where t is the step, and η is the learning rate. The learning rate is a hyperparameter for controlling how much the model's parameters will change, with respect to the estimated output error.

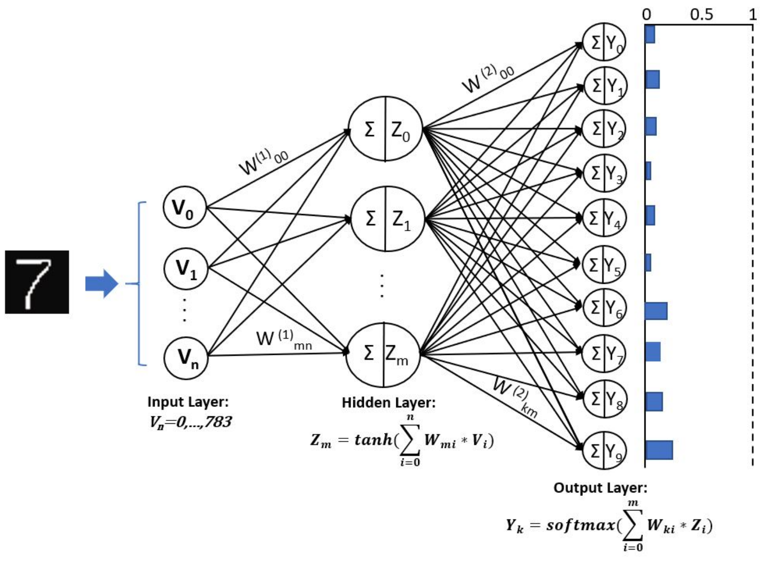
Before starting the training phase, the chosen dataset is divided into two subsets; the training and testing set. Training is done using the training set, whereas the network's accuracy is evaluated after training, using the testing set. In each epoch, the cost function is evaluated and the gradients with respect to the weights and biases are calculated (forward pass) for an input-output example. The estimated output error is fed into the network backwards and propagated layer by layer, in order to compute the corresponding gradients in each layer. The iterations of the algorithm stop, when the cost function is sufficiently minimized for a certain number of epochs. To avoid $overfitting$ [54], a method called validation-based early stopping [55] is used. In this method, the training set is further split into a training and validation set. After each epoch the accuracy is evaluated using the validation dataset. The training phase is stopped when the validation error is minimized. Note that this is not an exact algorithm as it does not necessarily find the global minimum of the cost function, for the given parameters of the network. Therefore, different combinations of the parameters can lead to the same, near the same or a better/worse local or a global minimum of the cost function.

2.1.1.2 Inference

After training, the network's parameters are optimized and fixed, and the network can be evaluated using new example data (inference). An example of a trained 2-layer NN is displayed in Figure 2.3b. The example image is fed into the first layer as a vector V_n ($n = 0, .., 783$), passes through the hidden layer by calculating the weighted sums and applying the activation functions, as displayed in figure 2.3. In the output layer the softmax function activates the corresponding neuron; showing the classification result, in this case number 7. However, in the case where the network is not trained, the softmax's output does not distinguish any class, as shown in Figure 2.3a.

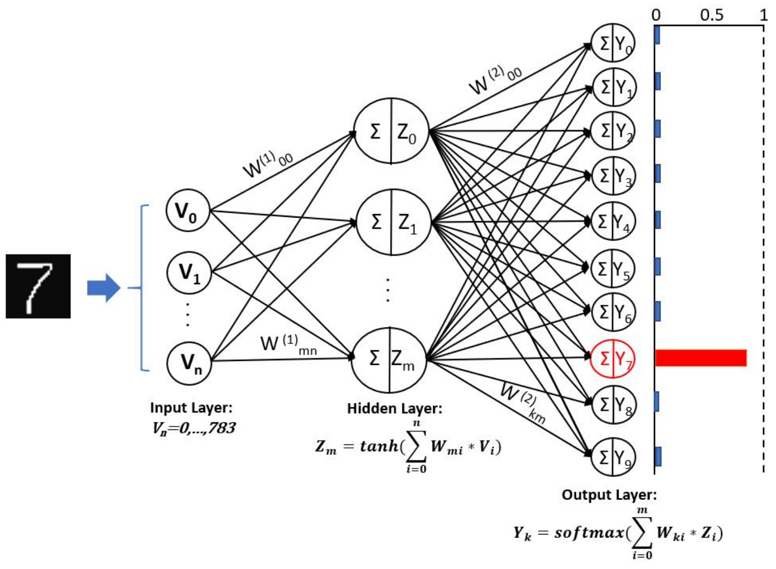
2.1.2 Neural network architectures

Neural networks are widely used for tackling a wide range of problems in different domains, for instance in image classification, speech translation [56], object recognition [57] and image segmentation [6]. There are several categories of NNs depending on their architecture, such as Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Binary Neural Networks (BNNs) etc. In this chapter, only relevant to this work NNs, are going to be discussed.



0
0.5
1
Input Layer:
V_n=0,...,783
Hidden Layer:
$Z_m = tanh(\sum_{i=0}^{n} W_{mi} * V_i)$
Output Layer:
$Y_k = softmax(\sum_{i=0}^{m} W_{ki} * Z_i)$

(a) Untrained network.



0
0.5
1
Input Layer:
V_n=0,...,783
Hidden Layer:
$Z_m = tanh(\sum_{i=0}^{n} W_{mi} * V_i)$
Output Layer:
$Y_k = softmax(\sum_{i=0}^{m} W_{ki} * Z_i)$

(b) Pretrained network.

Figure 2.3: Inference using a 2-layer NN for the MNIST dataset.

2.1.2.1 Deep Neural Networks (DNNs)

A neural network that has more than 1 hidden layer is called a Deep Neural Network (DNN). The name 'deep' refers to the depth of the network, deriving from the number (>1) of hidden layers. DNNs are used for tackling difficult classification problems as they can model complex non-linear relationships. They demonstrate intelligent behavior, based on the fact that a large number of neurons cascaded in multiple layers, thus being able to learn more complex features [46]. Apart from having only simple fully connected neurons as components, DNNs can also have different layers such as convolution layers, as described in the next subsection.

2.1.2.2 Convolutional Neural Networks (CNNs)

Convolutional neural networks is a category of NNs mainly used for image classification tasks [58]. CNNs are regularized versions of NNs and usually consist of convolutional layers, pooling layers and fully-connected layers. Contrary to NNs, where each neuron in one layer is connected to all neurons in the next layer, in convolutional layers only some neurons are connected to subsequent neurons of the network. This architectural feature adds sparsity to the network, as well as exploits the locality of the features in an image, i.e. that pixels closely together are more correlated than distant ones. Apart from utilizing spatial information, CNNs have also build-in invariant properties, which makes the network less prone to overfitting and a better approximator.

An example of a CNN is presented in Figure 2.4. The first part of the CNN's structure is responsible for learning and extracting the features from the corresponding example dataset (feature extractor), whereas the second part consisting of the fully-connected layers, is used for classifying the image in the correct category (classifier). As mentioned the network is usually composed of:

- Convolutional layers

  As the name indicates, convolutional layers perform a convolution operation between the input image and a filter (kernel). Depending on the weights of the filter, different features can be extracted in each convolution. By sliding the filter over the image, the dot product is computed, creating a feature map. The depth of the feature map is equal to the number of filters used in every convolutional layer. In order to add some non-linearity in the network, for learning non-linear data, convolutional layers are usually followed by an activation function, such as the ReLU function.

- Pooling layers

  Convolution layers are usually followed by a down-sampling layer, namely a pooling layer [59], for downgrading the created feature map, without loss of information. Different types of pooling layers can be applied, for example max, average, sum etc. The most common is max pooling, which takes the largest element from the created feature map, within a selected neighborhood.

- Fully-Connected (FC) layers

  A fully-connected layer is a traditional multi-layer perceptron, as described in section 2.1.1, followed by an activation function. Convolutional and pooling layers extract high-level features of the input image. Most of the extracted features might be suitable for getting a good classification result, however, FC layers help the network learn also combinations of those features.

CNNs are trained similarly to multi-layer perceptrons using the back-propagation algorithm. The main difference is that the filter weights have to also be optimized during training, for scoring a good classification result. As mentioned in section 2.1.1.1, first

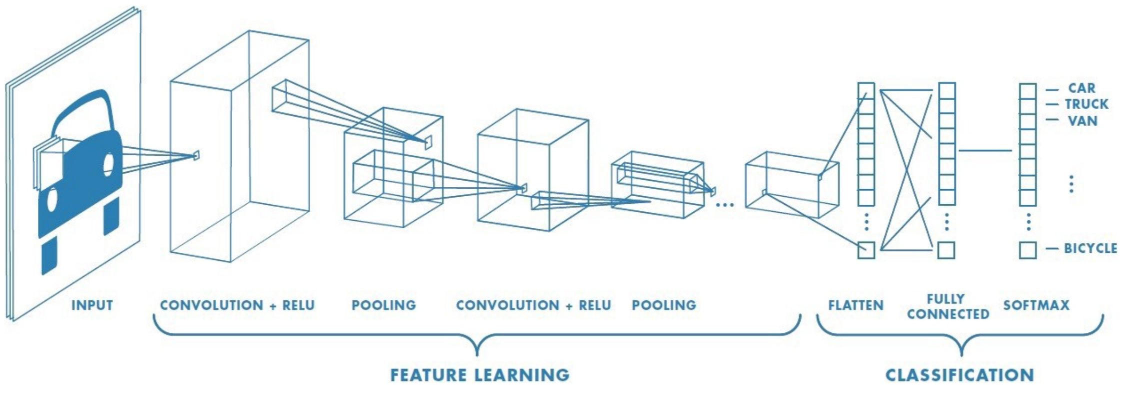


Figure 2.4: A Convolutional neural network architecture [5].

the gradients of the error concerning all weights in the network are calculated. Next, using the gradient descent method, we update all the weights and biases in such a way that the output error is minimized.

2.1.2.3 Binary Neural Networks (BNNs)

Although traditional deep neural networks have shown great success in solving a variety of problems, their computation and resource intensive nature, requires high storage, bandwidth and energy, in order to be used. For example, the VGG-16 network contains about 140 million 32-bit floating-point parameters, for classifying images using the ImageNet dataset [60]. The entire network occupies more than 500 megabytes of storage and performs $1.6{\times}10^{10}$ floating-point arithmetic operations. Thus, it is evident that DNNs need high-performance hardware platforms to be realized, which makes them unsuitable for embedded and battery-powered devices.

Bearing that in mind, researchers have proposed solutions for compressing and lightening the power-hungry nature of DNNs, namely parameter pruning [61] and quantization [62]. Quantization is a promising solution as it leverages the costly floating-point operations' problem, by representing the model's parameters with lower precision. An extreme quantization method is binarization, where data are 1-bit values, holding a value of +1 or -1. Using this method the weights and biases, as well as the activation functions in some cases, are binarized, represented by 1-bit utilizing less memory. Moreover, the high-cost matrix multiplication operations can be replaced with efficient bitwise XNOR and Bitcount operations, when 1-bit operands are used [37]. Therefore, it is clear that, Binary (or Binarized) Neural Networks (BNNs) as they are called, enjoy a number of hardware-favourable properties including memory savings, energy efficiency and considerable acceleration.

Due to this advantages, BNNs are suitable to be applied in various resource constrained computing platforms such as embedded devices. Thus, BNNs are the target NN architecture addressed in this thesis work. A significant number of works have been published investigating binary neural networks [63], some achieving near state-of-the-art accuracy compared to the corresponding floating-point implementations. Specifically, low-precision BNNs have been proposed [64], [65], [66], [36], suggesting that a precision

Work	BitWidth (W/A)	Architecture	Dataset	Accuracy (%)
[69]	32/32	VGG-small	CIFAR-10	93.8
BNN [64]	1/1			91.7
BinaryConnect [70]	1/32			89.9
[69]	32/32	ResNet-20		92.1
CI-BCNN [71]	1/1			91.1
[69]	32/32	ResNet-50	ImageNet	76
SYQ [72]	1/8			70.6
[69]	32/32	AlexNet		57.1
[73]	1/32			56.8
TSQ [74]	1/1			58

Table 2.2: Comparison of low-precision BNNs with their corresponding floating-point implementations.

of 8/16 bits is sufficient for training, while in inference the DNN's parameters require only 1 bit to be represented. Authors in [36] develop a low-cost implementation which has x32 less memory requirements, while being x58 faster with significant power savings, compared to the corresponding floating-point network. In Table 2.2 we evaluate the low-precision BNNs compared to their corresponding floating-point implementation. The bit-width (W/A) refers to how many bits are used to represent the weights (W) and activations (A). It is observed that in most cases low-precision NNs achieve high accuracy in the CIFAR-10 dataset, while the accuracy is slightly decreased for the ImageNet dataset depending on the NN implementation. Therefore, it is clear that BNNs are capable of having accuracy results as good as full-precision networks. Nonetheless, due to the quantazation of the weights and the activations, they suffer from a noticeable accuracy drop in more complex datasets, such as the ImageNet [63]. Several methods have been proposed for improving the quantazation error [67], the gradient error caused by the STE [68] as well as optimizing the loss function [36], resulting into better accuracy even for the ImageNet dataset.

As mentioned in [64], the first step in realizing BNNs is to decide how the weights and biases are to be binarized. Two methods are proposed:

- Deterministic:

$$x_{bin} = sign(x) = \begin{cases} -1, & \text{if } x \leq 0 \\ 1, & \text{otherwise} \end{cases} \tag{2.9}$$

- Stochastic:

$$x_{bin} = \begin{cases} +1, & \text{with probability p} = \sigma(\text{x}) \\ -1, & \text{with probability 1-p} \end{cases} \tag{2.10}$$

The σ(x) function refers to the 'hard Sigmoid' and equals: $\sigma(\text{x}) = \text{clip}(\frac{x+1}{2},\ 0,\ 1) = \max(0,\ \min(1,\ \frac{x+1}{2})$. Deterministic binarization is very straight forward and simple to implement, whereas stochastic binarization requires hardware to generate random bits when quantizing.

BNNs are trained using the back-propagation algorithm similarly to full-precision networks. However, the binarization function (e.g sign) is not always differentiable. Therefore, the gradient of the sign function is approximated using the straight-through estimator (STE) proposed by [75]. The STE is defined as:

$$clip(-1, 1) = max(-1, min(1, x))$$

Using the STE for estimating the gradient of the sign function enables training the network using back-propagation as in full-precision networks.

2.1.3 Applications of neural networks

Neural networks have gained phenomenal success in various applications domains; in computer vision tasks [58], speech recognition [56], natural language processing [76], and others. Although NNs can solve a variety of problems, some specialization, i.e. different architectures, is required for solving different tasks. Among others, DNNs are capable of solving very complex problems such as image segmentation (figure 2.5 [6]) and object recognition [57]. An emerging application that heavily depends on object recognition

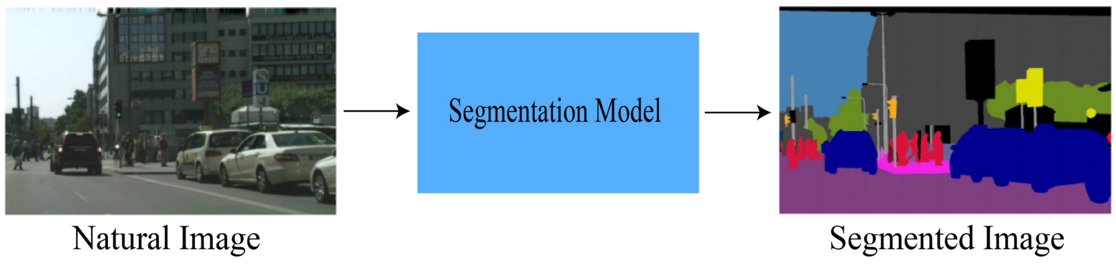


Figure 2.5: An example of image segmentation using DNNs [6].

and detection is autonomous driving [77]. Another application domain that DNNs are widely used is healthcare [78]. For example, CNNs using using deep learning are used for medical imaging, such as classifying brain tumors [79] or assisting in Alzheimer's disease detection [80].

2.1.4 Challenges of neural networks

Although NNs have shown great success in tackling various problems, they still have several limitations to be considered and overcome. Specifically, modern state-of-the-art DNNs require a huge consumption of computational resources, computation time, energy and memory. One reason for that is the DNNs' need for processing and storing huge amounts of data, in order for them to be trained correctly. A sufficient amount of training data is of critical importance, for the network to not overfit and obtain satisfying generalization behavior. For example, the well-known dataset ImageNet [81] has a whopping size of 150Gb. Another reason for the aforementioned limitations of DNNs is the big number of operations they require, in order to be trained (or infer).

Many state-of-the-art network architectures entail millions or billions of parameters to be optimized over thousands of epochs. For instance, DeepLab [82] , which is among the most popular image segmentation approaches, requires more than 6000 training epochs in order to tune millions of parameters.

In order to save computational resources, time, energy and memory, researchers have been trying to address the two aforementioned bottlenecks (big data processing and storing, complex and numerous operations). In particular, especially for low-power devices such as mobile phones, BNNs have been proposed [64, 65, 66, 36] (see Section 2.1.2.3), which remove the need for costly floating-point operations and require less memory in general. For example, in [36], a low-cost BNN implementation is developed, which has x32 less memory requirements, while being x58 faster and significant power savings, compared to the corresponding floating point network. Another solution that is currently being explored is the use of Computation-In-Memory (CIM) computer architectures for implementing neural networks, which severely reduce the amount of computations (e.g. the matrix-vector multiplication can be run with $O(1)$ complexity in certain CIM frameworks) and the memory hardware used. CIM is thoroughly discussed in the next section. In fact, the developments of the present thesis employ both BNNs and CIM. Finally, other solutions that have been proposed to alleviate the aforementioned bottlenecks (some of which can be combined with BNNs and/or CIM) include: a) hardware-specialized platforms for realizing DNNs, such as ASICs (Application-specific integrated circuits, e.g. [83]) and FPGAs (Field Programmable Gate Arrays, e.g. [84]), and b) utilizing powerful GPUs (General Purpose Units), which are known for their high degree of data parallelism and memory bandwidth [85] (although the slowdown that the single-GPU speed scaling progress is experiencing, has shifted the focus elsewhere).

2.2 Computation-In-Memory (CIM) Binary Neural Networks (BNNs)

Nowadays, most computing devices are based on the von-Neumman architecture paradigm. However, as mentioned in section 1.1, this architecture suffers from intensive data movement between the processor and the memory, resulting in high energy consumption and increased latency. This data bottleneck becomes even more critical in data-intensive applications such as neural networks as mentioned in the above section. Therefore, researchers have shifted their research into developing a new paradigm where data is processed where it makes sense; in the memory unit [22], [86], namely Computation-In-Memory (CIM). CIM architectures provide an efficient solution for implementing NN applications on hardware as it avoids the data movement and requires less power.

Among others, when designing a CIM based architecture one should consider three factors; the computation location, memory technology, and computation parallelism [21]. As far as the memory technology is concerned various works have proposed using traditional memory cells such as DRAMs and SRAMs for running CIM-based NNs on hardware [87]. Moreover, new emerging non-charge-based memories are also proposed, namely *memristors*, because of there favorable properties, such as non-volatility, zero static power consumption and small cell area [18]. Memristors consist of resistive memories (e.g. Resistive Random-Access Memory (ReRAM or RRAM) [88], Phase Change Memory (PCM) [89]) and magnetic memories (e.g. Magnetic RAM (MRAM) [90], Spin-transfer torque MRAM (SST-MRAM) [91]).

A CIM architecture consists of the memory array and the peripheral circuits. The crossbar architecture of the CIM memory array is shown in Figure 2.6b. The crossbar structure has n wordlines (input voltages). Each wordline and bitline is connected through a bit-cell (1T1R) memristor (a RRAM device in this case) at their intersection. CIM crossbar arrays can be used to efficiently perform Matrix-Vector Multiplication (MVM) operations. In Figure 2.6b a MVM operation can be performed, by applying a voltage vector $V{=}V_j$ (where $j{\in}\{1,n\}$) to the RRAM crossbar matrix of conductance values $G{=}G_{ij}$ (where $i{\in}\{1,m\}$, $j{\in}\{1,n\}$). At any instance, each column performs a Vector-Vector Multiplication (VVM) or a Multiply And Accumulate (MAC) operation, with the output current vector I, in which each element is $I_i{=}\Sigma V_j \cdot G_{ij}$. Note that all m MAC operations are performed in parallel with O(1) time complexity.

This O(1) MVM computation potential of CIM provides fast and efficient computing power for different neural network applications, such as BNNs. Figure 2.6a shows a simplified 2-layer BNN consisting of the input layer, hidden layer and output layer with fully-connected neurons.Each neuron calculates the weighted sum of the input vector and the weights, and then passes it through an activation function given in equation (2.11). From equation (2.11) it is observed that the weighted sum operation can be easily mapped to a RRAM based crossbar array shown in Figure 2.6b. In this case, the input vector V_j is applied as a voltage, while the weights (W_{ij}) are mapped on the corresponding conductances G_{ij} of the RRAM devices.

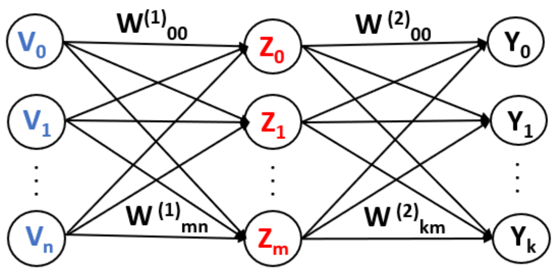
$$Z_i = f(\sum W_{ij} \cdot V_j) \tag{2.11}$$

where Z$_i$ is the output of the neuron i, f is the activation function, $\sum W_{ij} \cdot V_j$ is the weighted sum of the input values and their respective weights. Several works have been proposed for implementing BNNs on CIM memristive based architectures [37], [92]. For instance, in [37] authors propose a modified bit-cell RRAM-based crossbar array for BNNs, where the high-precision MAC operations are replaced by efficient XNOR and bit-counting operations.

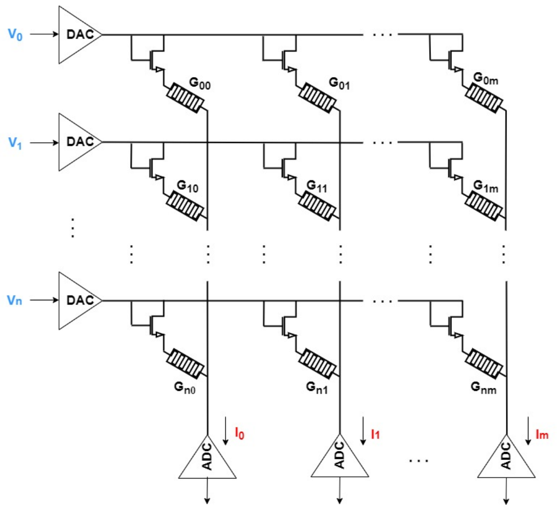
2.3 RRAM devices

2.3.1 Basics and working principle

Resistive random-access memory is a type of non-volatile memory, also called a memristor. The working principle of RRAM devices is based on the reversible formation or disruption of the Conductive Filament (CF) in the resistive layer leading to a high (HRS) or low resistance state (LRS). The structure of RRAM devices and I-V curve is presented in Figure 2.7. As shown in the figure, a RRAM device consists of a metallic oxide sandwiched between the two regions, i.e., the doped region (top electrode) and the undoped region (bottom electrode). The size of the CF determines the resistance state of the device. The typical read and write operations as in traditional memories can be applied.



(a) Demonstration of a 2-layer neural network mapped on a RRAM crossbar array.



(b) RRAM crossbar array structure, where V_n is the input voltage, G_{nm} the RRAM's conductance value and $I_m = \sum G_{nm} \cdot V_n$.

Figure 2.6: RRAM crossbar array for CIM-based 2-layer NN implementation.

When a sufficiently high positive voltage (higher than the set threshold voltage, V$_{set}$) is applied, some of the bonds between the ions break and form a conductive filament (CF) of vacancies (represented as white circles in Figure 2.8) that can conduct current. This increases the size of the CF, representing the low resistance state (LRS). On the contrary, when a negative voltage (lower than reset threshold voltage, V$_{reset}$) is applied, some ions move back into the oxide region as shown in Figure 2.8, thus reducing the size of the CF. As a result, the device has a high resistance state (HRS). Figures 2.7(b) and (c) show the I-V curve of an RRAM device when a unipolar and bipolar mode is used [7]. Briefly in unipolar switching, the RRAM device switches from a LRS to a HRS and vise-versa symmetrically when both a positive and negative voltage is applied. As a result, a set/reset can happen at the same polarity. Contrary, in bipolar switching the direction of the applied voltage determines the switching of the device. In order to read the state of the RRAM device, an appropriate current is applied on the top electrode resulting to a current that is read out by the corresponding periphery circuit.

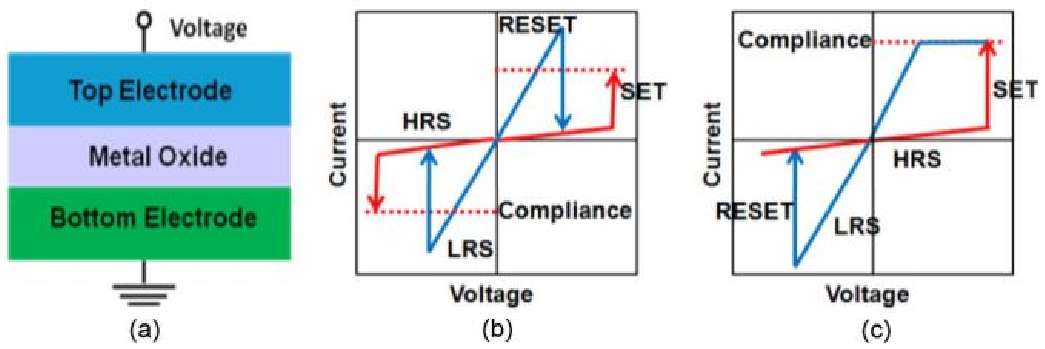


Figure 2.7: Schematic of MIM structure for metal–oxide RRAM, and schematic of metal–oxide memory's I–V curves, showing two modes of operation: (b) unipolar and (c) bipolar [7].

Memory	**SRAM**	**DRAM**	**NAND Flash**	**PCM**	**STT-MRAM**	**RRAM**
Cell area	$>100F^2$	$6F^2$	$<4F^2(3D)$	4-20 F^2	6-20 F^2	<4 F^2
Cell element	1T1C	1T	1T	1T(D)1R	1T1R	1T(D)1R
Read time	~1ns	~10ns	~10s	<10ns	<10ns	<10ns
Write time	~1ns	~10ns	~100μs-1ms	~50ns	<5ns	<10ns
Write energy (per bit)	~fJ	~10fJ	~10fJ	~10pJ	~0.1pJ	~0.1pJ
Retention	-	~64 ms	>10years	>10years	>10years	>10years
Endurance	$>10^{16}$	$>10^{16}$	$>10^4$	$>10^9$	$>10^{15}$	$\sim10^6 - 10^{12}$
Non-volatility	No	No	Yes	Yes	Yes	Yes

Table 2.3: A comparison of different memory technologies [94].

2.3.2 Applications and benefits

RRAM devices have recently gained widespread attention due to their non-volatility, high integration density and their ability to overcome memory bandwidth issues by executing operations within the memory [93]. A comparison of RRAM devices with other memory technologies is presented in table 2.3. Great advantages of RRAM devices include their non-volatility, resulting into zero static power, small cell area, leading into high density integration and scalability, and long lasting retention properties. Compared to traditional non-volatile memories, e.g. the NAND Flash, RRAMs are high-speed devices as they have smaller read and write time values. These properties make RRAM attractive for various applications ranging from non-volatile memory [95], logic based around computation in memory [14, 96, 97, 98, 23], implementations of physical unclonable function (PUF) for hardware security applications [99] and neuromorphic computing [22], [31], [37].

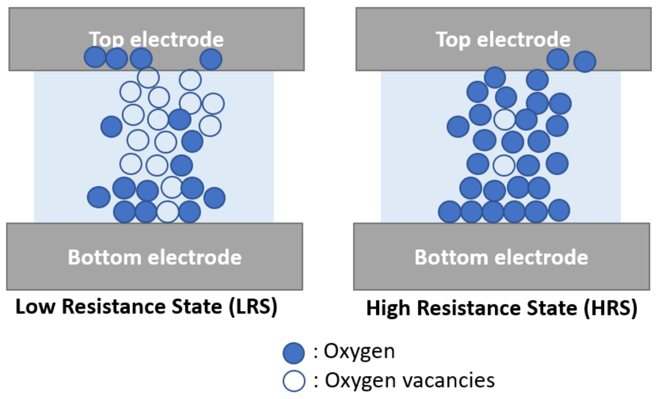


Figure 2.8: Structure of a RRAM device programmed in a low and high resistance state.

2.4 Defects in RRAM devices

Despite their significant advantages RRAM devices suffer from various reliability and variability problems, hindering their large scale manufacturing. RRAM defects can be broadly divided into **permanent** defects, defects at t=0 due to fabrication challenge or **transient** defects at t>0 due to device variability and runtime issues. During operational time, however, transient defects can also become permanent defects.

2.4.1 Permanent defects

In the case of a permanent defect, the RRAM device is stuck in either a high or low resistance state. The RRAM device's value can not be tuned anymore with a write operation. Root causes include overforming of the device [100] and open defects [7], such as a permanent open switch. Apart from permanent defects originated from the manufacturing stage, transient defects can also become permanent. For instance, multiple read or write operations can disturb the state of the RRAM device making it in the course of time, get stuck at a low or high resistance state [7]. Therefore, permanent defects include both defects induced by the manufacturing stage as well as run-time generated issues that turn into permanent defects.

2.4.2 Transient defects

On the other hand, transient defects are dynamic, meaning that the state of the RRAM device is not fixed in a particular resistance value, but can be tuned and changed from time to time during run-time. These intermittent problems are linked to the inherent properties of RRAM devices, attributed to the stochastic nature of the oxygen vacancies and ions processes [7]. Therefore, process variation is a significant root cause of transient defects. Variation makes the device's resistance value differ from its ideal value and can be cycle-to-cycle (C2C) or device-to-device (D2D). D2D variation occurs when different RRAM devices show different resistance characteristics under identical programming conditions [27], while C2C variation happen when a single RRAM device shows different

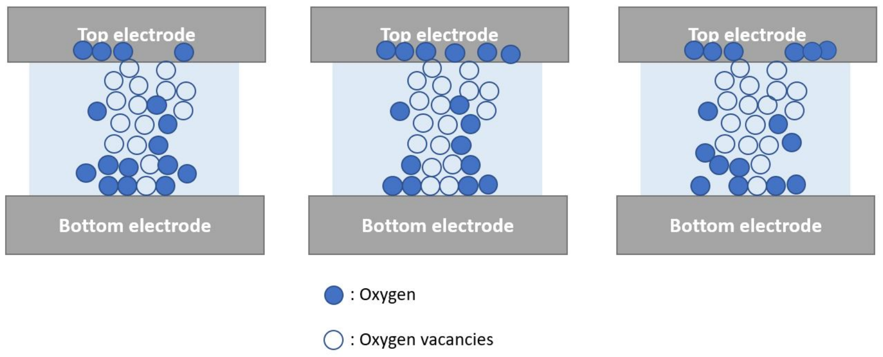


Figure 2.9: All 3 RRAM devices are programmed to have the same resistance value. Due to the random formation/disruption of the conductive filament, each device has a different resistance value.

resistance characteristics from time to time [101]. A visual example of D2D variation is presented in Figure 2.9. Aside from variation, the accumulated effect of large numbers of read/write operations can lead to significant change (drift) of the resistance state of RRAM devices [28], which can be considered a transient defect. At last, the non-linear and asymmetric I-V characteristics of resistive memories (shown in Figure 2.8) causes the RRAM device to deviate from its ideal value. For example, variation in the read voltage can lead to different effective resistance ratios, causing functional errors [29].

2.5 RRAM defect modeling

RRAM defect modeling provides a layer of abstraction, for understanding the behavior and testing easier RRAM devices. Permanent defects arising from the manufacturing stage and can be described by the Stuck-at-fault (SAF) model. When a device is forced to a permanent low state, suffers from a Stuck-at-One (SA-1) fault, whereas a device stuck at a high resistance value, is described by a Stuck-at-Zero (SA-0) fault. Apart from permanent faults originated from the manufacturing stage, SAFs can also be a result of transient defects. For instance, multiple read or write operations can disturb the state of the RRAM device making it in the course of time, get stuck at a low or high resistance state [9]. As far as transient defects are concerned, variability issues as well as resistance drift as described in section 2.4, can lead to an undefined state fault. Aside from variation, a transient fault might be induced by a read or write operation. Read disturbance is a phenomenon in which a RRAM device switches its state during the read operation and depends mainly on the stress time and the initial resistance state of the device [30]. The latter can be modeled as a read disturb fault.

3 Related works

Although CIM is a fast and efficient computing paradigm for implementing neural networks on hardware, the RRAM fabrication and run-time induced reliability problems described in section 2.4, have significant impact on the inference accuracy of CIM-based neural networks. In particular, authors in [33], [32] investigate the effect of SAFs using a simple 2-layer and 3-layer NN for the MNIST dataset. Both works highlight the negative impact of SAFs on the classification accuracy, [32] states that inserting 10% of SAFs results in an accuracy drop of around 37%. Intermittent faults, as described in [11], [8], [102], such as C2C variation, data retention and the asymmetry of conductance tuning, play a key role in the degradation of the classification accuracy as well.

Aside from RRAM device related non-idealities, the crossbar array also suffers from non-ideal characteristics. Usually, along with the crossbar array, peripheral circuits are required such as Digital-to-Analog Converters (DACs) and Analog-to-Digital Converters (ADCs). Due to the analog nature of computing, several non-idealities can lead to errors in the MVM computations [100]. Crossbar array non-idealities can derive from source, sink and wire resistances as well as access devices, selectors and sneak paths. Regarding the periphery, a key source of non-idealities is the ADCs and DACs [103]. Authors in [103], show that larger DNNs suffer more from non-idealities, for instance the Resnet-50 for the ImageNet dataset, shown an accuracy degradation of 32% when it was executed on non-ideal crossbar arrays.

Therefore, it is evident that a fault tolerant neural network design is crucial for restoring the classification accuracy, paving the way for unlocking the full potential of CIM based neural networks. In this chapter, the state-of-the-art fault-tolerance techniques are presented, focusing on RRAM-based solutions. We divide these techniques into two categories: software and hardware based. However, it is noted that this categorization is not exclusive, as some proposed fault-tolerance solutions combine multiple approaches for having better results. Thus, the classification of the fault-tolerance techniques is done, by considering what is the main contribution of each work.

3.1 Software-based solutions

RRAM reliability issues have significant impact on the inference accuracy of NNs mapped to RRAM-based crossbar arrays, making the need of finding fault-tolerance techniques essential. In this respect, software-based solutions have been developed, which can further be classified based on their proposed solution into retraining and remapping methods. It is noted that some works combine both retraining as well as remapping techniques in order to further restore the accuracy.

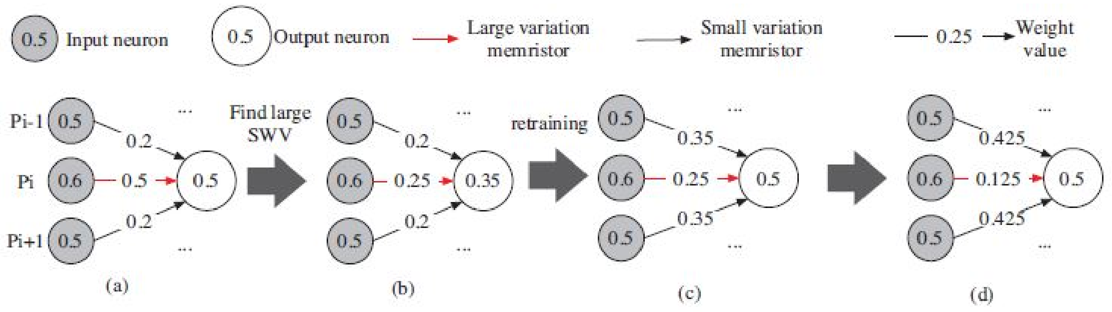


Figure 3.1: Weight changing in the neural-network retraining method: (a) pre-trained weight; (b) fixing the weight connection; (c) after retraining; (d) in the next iteration [8].

3.1.1 Retraining

Authors in [8], [32], [9] investigate the self-healing capabilities of the network to tolerate SAFs and variations, via a modified re-training method. Works [8], [32] are off-line retraining methods as they are applied once after the manufacturing stage, while the work in [9] proposes an on-line retraining framework for tolerating RRAM faults during run-time. An important step in the retraining method is the extraction of the RRAM fault distribution. The failure map of the RRAM devices is obtained by utilizing a quiescent voltage comparison method in [9], while in [32] the location of the defective devices is acquired by chip testing.

Authors in [8] evaluate their proposed method by using a 2-layer NN on the MNIST dataset. The network's accuracy is restored by incorporating weight variations into the retraining phase. In particular, weights that are mapped on RRAM devices with high variability are gradually decreased, while non-defective weights in the near neighborhood are increased (figure 3.1). For a low percentage of 10% SAFs and high variations, the retraining technique recovers the accuracy up to 71%, when the original accuracy is 90%. The second offline retraining method proposed by [32], first finds the significant and insignificant weights and evaluates the impact of SAFs on the inference accuracy. After, the fault distribution of SAFs is extracted from chip testing. This fault map is used as a constraint for optimizing weights that are mapped on non-defective RRAM devices. For a fault rates up to 20% the accuracy is restored by 98.1% when a 2-layer NN on the MNIST is evaluated.

The on-line fault-tolerance framework of [9] (figure 3.2) focuses at enabling more accurate on-chip training. The framework consists of a fault-detection phase and a fault-threshold training technique. If the classification accuracy is degraded sufficiently, the fault-detection phase is used to identify the new fault map. Consequently, the fault-threshold training technique is used to retrain the network. The threshold-training method updates the weights only when a large update (relative to the threshold value) is needed. Therefore, this technique aims at mitigating SAFs caused by the limited life endurance of RRAM devices. Overall, the fault-tolerance framework improves the lifetime of RRAMs by 15 times.

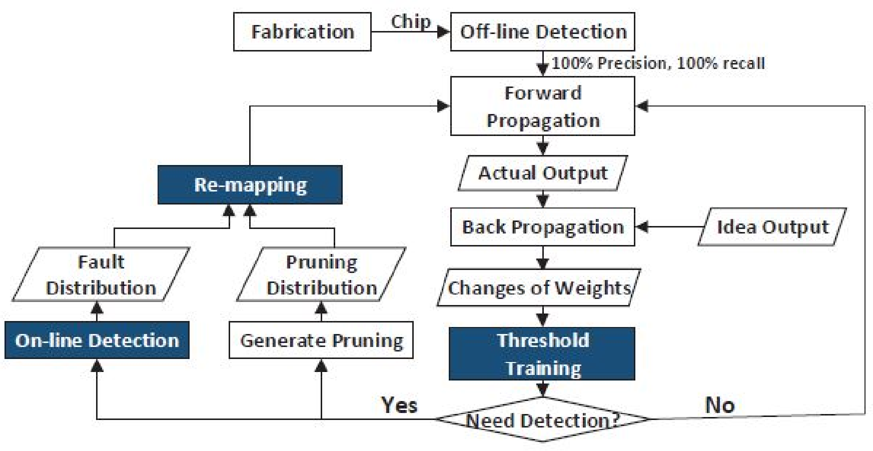


Figure 3.2: The fault-tolerant training method with (1) a threshold-training method to reduce the write workload, (2) an online fault detection phase to detect fault locations, and (3) a remapping phase to avoid or reuse faulty cells. [9].

3.1.2 Remapping

Remapping algorithms aim at finding the optimal mapping between the RRAM devices and the corresponding weights, for reducing the mapping error. Such proposals have been widely investigated in recent literature, including row and column matching between weights and RRAM devices [8], [33] and approximation of weights with multiple RRAM devices [31], [102]. Authors in [33], propose three matrix transformations for fault-tolerance; a row flipping transformation, a matrix permutation and a value range transformation. The row flipping transformation enlarges the mapping space by flipping SA0 weights into SA1 weights and vise versa. In the matrix permutation, neurons are permuted for mapping small (large) weights to RRAM devices with small (large) conductance values in the crossbar array. A different approach is proposed in [9] where the inherent sparsity of a neural network is exploited to tolerant SA0 faults in RRAM cells. This is done by re-ordering the columns/rows of the weight matrix, resulting in mapping the zeros in the weight matrices to RRAM devices with SA0 faults. The last work mentioned in this section is a Mapping Algorithm with inner fault-tolerance (MAO) [31]. MAO uses at least 2 RRAM devices {G+, G-} for representing positive and negative weights. RRAM devices {G+, G-} are initialized with G_{max} and G_{min} respectively. Then MAO tries to find the best pair of {G+, G-} for each weight, which minimizes the mapping error. Without using any further fault-tolerance techniques MAO improves the accuracy slightly from about 20% to 40% for the MNIST dataset. For restoring the accuracy even further authors propose using redundant devices for enlarging even more the mapping space.

3.2 Hardware-based solutions

3.3 Redundancy

The hardware-based solutions utilize redundancy schemes to improve fault-tolerance [102], [41], [38]. Redundancy schemes usually involve using extra rows, columns and whole crossbar arrays for remapping. In [41], a distribution-aware and re-configurable redundancy scheme combined with MAO is proposed, using redundant columns or crossbar arrays, for tolerating SAFs. The number of redundant columns or crossbars is determined when the fault distribution is being known (can be directly post-fabrication or during run-time). When SAFs follow a non-uniform and unknown distribution the proposed solution reduces the number of redundant RRAM cells from more than 200% to less than 40% and 60%, respectively. When redundant columns are used, the recognition accuracy of the MNIST dataset is recovered close to the ideal accuracy value, with an energy overhead of 37.58%. In [38], a fault tolerance technique for BNNs mapped on RRAM-based crossbars is proposed. First a March-type test [104] is used to obtain the fault map of the crossbar array, followed by a modified 4T1R (4transistor-1-Resistor) architecture for tolerating random SAF distributions. As the authors state, this re-configurable RRAM cell guarantees 100% fault tolerance for up to 50% fault density, when the MNIST dataset is considered.

3.4 Limitations of the state-of-the-art

Although, the aforementioned fault-tolerance solutions can improve the accuracy close to the ideal or to a great extent, their implementations still impose significant hardware and/or software overheads.

3.4.1 Limitations of software-based solutions

Software overhead related to the retraining techniques is mainly because of the extra retraining time. In addition, if faults cluster in a neighborhood in the weight matrix, the accuracy cannot be restored back to the ideal value with the existing solutions. In this case redundant devices are utilized by [8], [32], for recovering the accuracy closer to the fault-free value. Another limitation is that the fault distribution of the crossbar arrays should be known a-priori. If the fault map is acquired offline, then the retraining is performed only once after fabrication. This has the disadvantage that faults might arise during runtime which won't be considered in the retraining technique, resulting into accuracy degradation of the mapped network. If online training is considered as in [9], a fault detection mechanism should be developed, adding extra hardware and software overhead. As far as the remapping algorithms are concerned, their main limitation is the limited flexibility in matching between the RRAM devices and the weights. So far all solutions allocate significant software resources in finding the optimal mapping, which is proven to be a hard problem to be solved. This is mainly because the mapping error plays a crucial role in obtaining a good classification accuracy of the mapped NNs. While [9], which proposes a threshold retraining method and a remapping scheme, increases the lifetime of the RRAM devices, it has significant software overhead for the remapping algorithm and the fault detection phase. Overall, both approaches address limited faults and rely on hardware redundancy to restore the accuracy when the fault rate increases.

3.4.2 Limitations of hardware-based solutions

Redundancy schemes can indeed restore the classification accuracy of NNs mapped on RRAM based crossbar arrays, yet their main drawback is the high hardware and energy overheads. Moreover, there is no guarantee that redundant devices are not faulty as well as the primary ones, nullifying the benefit of remapping solutions in some cases. The solution of [41] has an extra challenge during manufacturing stage. Due to the fact that different number of RRAM devices are realized per column, depending on the fault-distribution, the crossbar array requires also extra routing and modifications. A similar challenge derives from the 4T1R architecture proposed in [38], where the array is also modified.

Therefore, an efficient and cost-effective solution, tackling the software (mainly caused by the mapping complexity) as well as the hardware overhead, as proposed in this paper, is crucial in order to reliably map NNs to RRAM-based crossbar arrays.

RRAM Fault Impact Analysis 4

In this section we analyze the impact of RRAM defects on the inference accuracy of BNNs. In the following analysis, the impact of permanent faults (Stuck-at-fault (SAF)) and intermittent faults (conductance variation) are considered, as they can aggregate the impact of other RRAM defects as explained in section 2.4. First, the architectural implementation and simulation framework is discussed. Next, the impact of SAFs and variation on the BNNs' accuracy is examined for various architectures. Lastly, overall conclusions are drawn regarding the fault impact analysis.

4.1 Evaluation setup

The evaluation setup is divided into two subsections: the architectural implementation and the simulation framework. The first subsection, presents how the chosen neural networks are realized on RRAM-based CIM crossbar arrays. Next, the simulation framework is introduced, which includes the different neural network architectures and datasets chosen for evaluation, as well as how the considered permanent and intermittent faults, are modeled and simulated.

4.1.1 Architectural implementation

Two aspects are important in order to implement BNNs on RRAM-based CIM crossbar arrays: a) the structure of the RRAM crossbar array and b) the NN architecture. Regarding the first point, in this work we consider a binary One-Transistor one-Resistor (1T1R) bit-cell design, in which the cell can hold 2 states; HRS representing the logic '0' and LRS representing the logic '1'. A great advantage of binary RRAM devices is that they are easier to realize and program than multilevel RRAM, making them also less prone to manufacturing and operational faults [105].

A specific category of neural networks, BNNs, is selected to study the impact of RRAM faults and demonstrate the effectiveness of the proposed fault-tolerance techniques. As described in Section 2.1.2.3, BNNs use weights that can have two values; ± 1, utilizing only 1bit. The latter matches perfectly with the architecture implementation of having only 2 states per RRAM device. Since BNNs use binary weights, they can easily be mapped to RRAM-based crossbar arrays, as each weight can be stored as a

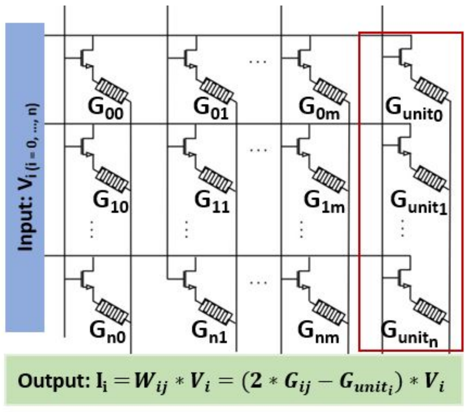


Figure 4.1: Implementation of selected mapping [10] on RRAM based crossbar array. The red frame shows the added extra column (G_{unit_i}), for realizing weights = -1.

conductance value in one RRAM device.

$$\underbrace{\begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix}^{\top}}_{\text{weights}} \cdot \underbrace{\begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix}}_{inputs} = 2 \cdot \underbrace{\begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_n \end{bmatrix}^{\top}}_{conductances} \cdot \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} - \underbrace{\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}^{\top}}_{col\_ones} \cdot \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} \tag{4.1}$$

$$W_i = \begin{cases} -1, & \text{if } G_i = 0 \text{ (HRS)} \\ 1, & \text{if } G_i = 1 \text{ (LRS)} \end{cases} \tag{4.2}$$

It is evident though that a one-to-one mapping of the BNN's weights ($W_i = \pm 1$) to the conductance values ($G_i = \{0, 1\}$) is not feasible as the value of -1 has to be stored in an RRAM cell. For this purpose, we use the mapping scheme shown in equation (4.1) [10]. This mapping is implemented by adding an extra column of RRAM cells (*col_ones*) in the crossbar array as shown in Figure 4.1. The *col_ones* holds unit conductance values ($G_{unit_i} = 1$). We take the weighted output sum of the crossbar array multiplied by 2, (i.e. the term $2 \cdot G_i \cdot V_i$ in equation 4.1) and subtract from it the output of the extra column ($G_{unit_i} \cdot V_i$). This scheme allows us to map both -1 and +1 weight values to the RRAM conductance states. The selected mapping technique has the advantage of reduced hardware overhead, compared to other methods, such as in [8], where 2 crossbar arrays are utilized for holding positive and negative values for the weights.

4.1.2 Simulation framework

The flow diagram of the simulation framework is presented in figure 4.2. First, the BNN models are trained using the widely used python packages Pytorch [106] and TensorFlow [107]. Several BNN architectures are evaluated for different datasets, for the sake of generality and variety of test networks. Once the weights are optimized in the training

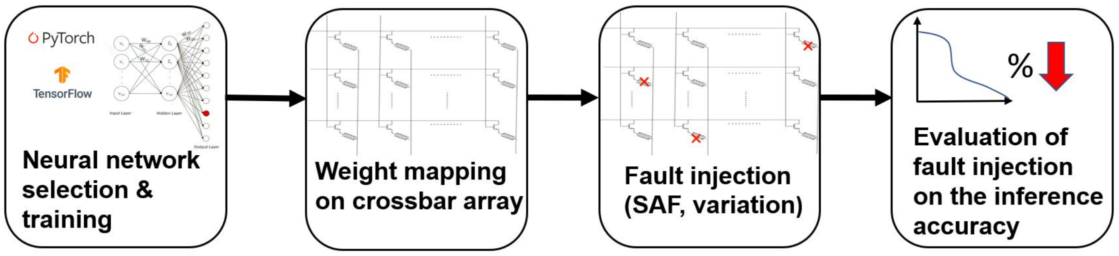

PyTorch
TensorFlow
Neural network selection & training
Weight mapping on crossbar array
Fault injection (SAF, variation)
%
Evaluation of fault injection on the inference accuracy

Figure 4.2: Simulation framework for evaluating the impact of RRAM faults on the BNN's accuracy.

#Layer	**Activation function**
BinarizeLinear(num_inputs=784, num_neurons=784)	Tanh
BatchNorm1d(num_neurons=784)	-
BinarizeLinear(num_neurons=784, 10)	Softmax

Table 4.1: Architecture of a 2-layer BNN for the MNIST dataset.

phase, they are mapped on the RRAM-based crossbar array using the aforementioned mapping algorithm. Next, in order to simulate SAFs and/or conductance variation, a certain portion of the ideal mapped weights are transformed to defective ones, by either fixing their values to -1 or +1 (SAF simulation) and/or adding some variation to the weights' values (conductance variation simulation). Finally, the defective weight matrices are used for evaluating the inference accuracy of the network.

For the sake of generality and variety, in this simulation framework different network architectures are evaluated. In particular, an n-layer BNN (n = 2, 3, 4), a Floating Point 2 Binary Network (FP2BIN) NN and a VGG BNN are used along with the MNIST [39], Fashion-MNIST [40] and CIFAR-10 [42] datasets. More details regarding the architecture of the BNNs can be seen in Table 4.1 and Table 4.2. The original inference accuracy of the networks is displayed in Table 4.3. Details for each dataset and its corresponding architectures, as well as more information for the fault modeling are presented below.

- **MNIST dataset**
  - This dataset consists of 70.000 28x28 gray-scale images, which represent handwritten decimal digits from 0 to 9. The original set is split to: 60.000 images for training and 10.000 images for testing.
  - Four different NN architectures are trained, namely FP2BIN and n-layer BNN with n = 2, 3, 4. The n-layer BNNs are modified versions of the BNN architecture proposed in [64]. The FP2BIN is trained using floating point weights like traditional multi-layer NNs, but after training and before inference, the weights are binarized and obtain values of -1 or +1. In all 4 architectures the binarization is carried out using the deterministic function mentioned in section 2.1.2.3, which is simpler to implement on hardware.
  - The number of neurons as well as the number of layers used in the network

#Layer	Layer type	Details	Activation function
1	C	BinarizeConv2d(kernel = 3x3, stride = 1, filters = 128)	-
	B	BatchNorm2d(128)	HardTanh
2	C	BinarizeConv2d(kernel = 3x3, stride = 1, filters = 384)	-
	P	MaxPool2d(kernel = 2x2, stride=2)	-
	B	BatchNorm2d(784)	HardTanh
3	C	BinarizeConv2d(kernel = 3x3, stride=1, filters = 768)	-
	P	MaxPool2d(kernel = 2x2, stride = 2)	-
	B	BatchNorm2d(768)	HardTanh
4	C	BinarizeConv2d(kernel = 3x3, stride=1, filters = 768)	-
	B	BatchNorm2d(768)	HardTanh
5	C	BinarizeConv2d(kernel=3x3, stride = 1, filters = 1536)	-
	B	BatchNorm2d(1536)	HardTanh
6	C	BinarizeConv2d(kernel=3x3, stride = 1, filters = 512)	-
	P	MaxPool2d(kernel = 2x2, stride = 2)	-
	B	BatchNorm2d(512)	HardTanh
7	FC	BinarizeLinear(8192, 1024)	-
	B	BatchNorm1d(1024)	HardTanh
8	FC	BinarizeLinear(1024,1024)	-
	B	BatchNorm1d(1024)	HardTanh
9	FC	BinarizeLinear(1024,10)	-
	B	BatchNorm(10)	LogSoftmax

Table 4.2: Architecture of the VGG BNN for the CIFAR-10 dataset, where layer type is: C=Convolutional, B=Batch normalization, P=Pooling, FC=fully-connected.

architectures are optimized for having higher classification accuracy. In particular, figure 4.3 shows the accuracy when altering the number of neurons for the n-layer BNN. It is observed that the highest accuracy (97.5 %) is reached with the 4-layer BNN with 784 neurons per layer, while the 3-layer BNN with the same amount of neurons per layers is very close with an accuracy value of 97%. For the 2-layer BNN the accuracy increases slightly, only about 0.5%, when more neurons are used, resulting to an accuracy of 96%, for 2048 neurons per layer. Thus, taking into account the trade-off between the obtained accuracy and the number of neurons, we employed BNNs with 784 neurons per layer, for the experiments over the MNIST dataset. Nonetheless, it must be noted that our goal is not necessarily having the most optimized architecture for a given dataset, but mainly using a well-trained NN, which has sufficient accuracy and can be easily implemented on crossbar arrays.

- **Fashion-MNIST dataset**

  The Fashion-MNIST is the second dataset used for evaluating the BNNs' accuracy. It contains Zalando's article images with a set of 60.000 training examples and a test set of 10.000 example images. Each 28x28 image belongs to one of the 10 following classes: t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. In this case, the highest accuracy (88%) is achieved with a 4-layer BNN with 784 neurons per layer. This 4-layer BNN is the only one used for experiments over the Fashion-MNIST dataset.

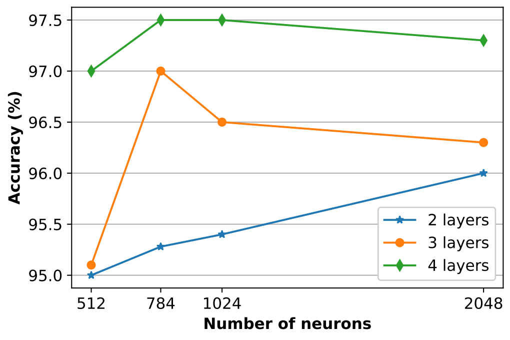


Figure 4.3: Investigating the accuracy behavior with respect to the number of neurons, for the n-layer BNN (n = 2, 3, 4) on the MNIST dataset.

- **CIFAR-10 dataset**

  The CIFAR-10 dataset consists of 60.000 32x32 colour images, divided into 50.000 training and 10.000 test example images. Each image is classified in 1 of the following 10 classes: aeroplane, automobile, bird, cat, dog, frog, horse, ship and truck. Compared to MNIST and Fashion-MNIST, it is a much more complex dataset to be learned, requiring a deeper network. Therefore, a modified VGG BNN architecture is used [64]. The VGG BNN has 9 layers (6 convolutional and 3 fully-connected) as shown in table 4.2. The first part of the network, which includes the convolution layers, is responsible for extracting the relevant features of the given dataset, while the fully-connected layers are mainly used to classify the extracted features and combinations of the features, for each example. All filters have a kernel size of 3x3 and, depending on the layer, different number of filters are used. The VGG BNN has an original classification accuracy of 90.09%, as shown in table 4.3.

- **Fault modeling**

  In this analysis, both permanent as well as intermittent faults are considered. Permanent faults are represented by the SAF model, and include defects generated from the manufacturing phase as well as intermittent faults that become permanent during run-time (described in section 2.4). SAFs are simulated by inserting them randomly in the weights' matrix. In particular, SA0s (HRS) are inserted as -1s and SA1s (LRS) as 1s, according to the aforementioned mapping. The number of SAFs injected into the network is equal to the considered percentage of SAFs multiplied by the total number of weights.

  Regarding intermittent faults, we consider the conductance drift and the conductance variation. In this investigation only device-to-device variations are considered. Device-to-device variations are simulated by replacing the ideal weights with a random value selected from a normal distribution as shown below:

$$weight \sim \mathcal{N}(\mu, \sigma^2),$$

Architecture	**# Layers**	**Dataset**	**Baseline Accuracy (%)**
FP2BIN	2	MNIST	90
n-layer BNN (n = 2, 3, 4)	2-4	MNIST	95.28, 97.1, 97.8
4-layer BNN	4	Fashion-MNIST	88
VGG BNN	9	CIFAR-10	90.09

Table 4.3: Evaluation Setup: BNNs and datasets.

where:

$$\mu = \begin{cases} -1, & \text{if } weight_{original} = -1 \\ 1, & \text{if } weight_{original} = 1 \end{cases}$$

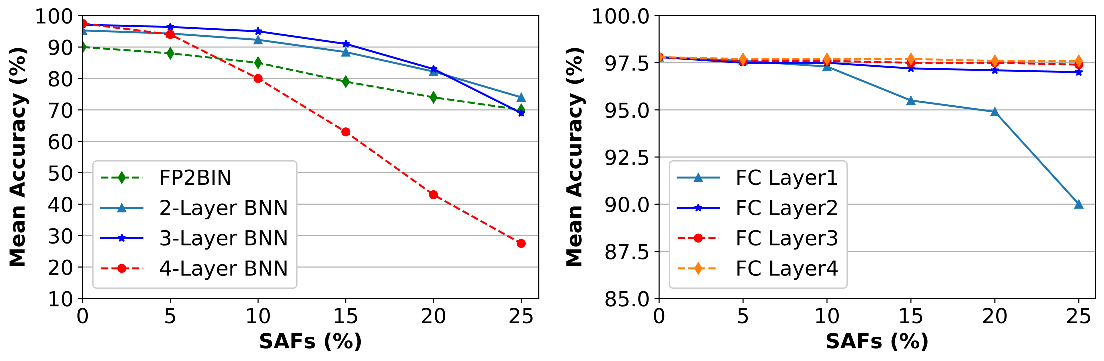
and $weight_{original}$ represents the value of the corresponding weight without the presence of conductance drift or variation. The conductance drift and variation is controlled by altering the σ value of the normal distribution. All fault simulations are evaluated over 500-1000 random samples using the Monte Carlo method, in order to obtain a statistical estimate on the inference accuracy.

4.2 SAF impact evaluation

Based on results obtained from chip testing, about 10% of SAFs are observed in a real RRAM crossbar chip [104]. Therefore, the percentage of inserted SAFs is considered to be up to 25%, in all test cases, including manufacturing as well as run-time produced SAFs. The impact of SAFs is evaluated on different BNN architectures and datasets along with different percentage ratios of SA0s and/or SA1s. In particular, only SA0s, only SA1s and a random combination of SA0s and SA1s are simulated. Moreover, a layer-wise fault analysis is conducted in order to determine how injecting faults in different layers of the network affects the classification accuracy. In the rest of the section the simulation results are presented.

4.2.1 Evaluation of n-layer BNNs

In the following figures the assumption is made that both SA0s and SA1s are injected randomly, unless stated otherwise, similarly to [32]. Figure 4.4a shows the impact of SAFs on shallow and deeper networks, for four different architectures of BNNs, using the MNIST dataset. All BNNs can tolerate faults up to 5% with minimal accuracy degradation. However, from the figure we can observe that the SAF-induced accuracy reduction increases with increase in the network depth, i.e. the more hidden layers, the higher the SAF impact on the network inference accuracy. The latter is clearly evident in the BNN architecture with 4 layers, where the accuracy drops to around 30% when 25% of SAFs are inserted. Regarding the 2-layer BNN the accuracy starts to drop evidently after inserting more than 10% of SAFs, going down to around 65%, whereas for the 3-layer BNN, the accuracy degradation is much sharper, reaching around 40% for



(a) Deep vs shallow network.

(b) Layer-wise SAF analysis.

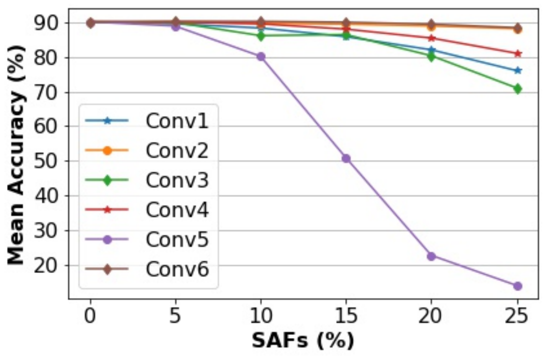
Figure 4.4: SAF impact evaluation on multi-layer BNN architectures for the MNIST dataset.

SAFs = 25%. This accuracy behavior is mainly due to the fact that the layers are fully connected, which enables faults to affect more neurons, ultimately severely harming the accuracy of deeper BNNs. Therefore, we can safely conclude that the 2-layer BNN is a relatively resilient architecture to SAFs compared to the deeper BNN variants.

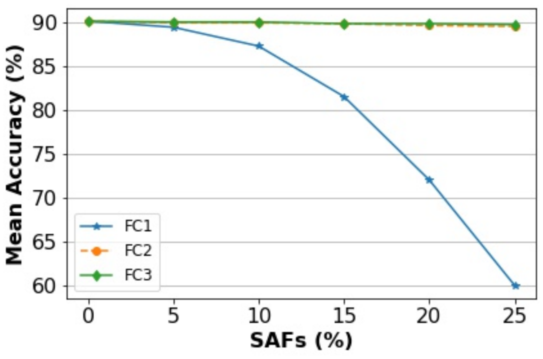
4.2.2 Layer-wise investigation

The impact of SAFs injected in a layer-wise manner is also investigated in this work as shown in Figures 4.4b and 4.5b. Figure 4.4b shows the accuracy reduction of a 4-layer BNN, by applying all the faults in a single layer at a time. It is observed that faults injected in earlier layers led to higher accuracy reduction, than faults in later stages. In particular, for a 25 percentage of SAFs inserted only in layer 1, the accuracy falls down by 8%, while when faults are injected only in layer3 or layer4 there is a minimal accuracy drop of about 1%. This could be potentially due to two factors: 1) faults in earlier stages can affect more features (e.g., in the first layer), 2) the impact of the faults can propagate to later stages leading to higher accuracy reduction, while faults in later stages (e.g., layer 4) affect fewer neurons.

A similar behavior is observed in figure 4.5b for the VGG BNN, especially for the fully-connected (FC) layers. FC layer 1 is mostly affected by the insertion of SAF, while the accuracy is almost left intact when faults are injected in FC layer 2, 3, 4. Regarding the convolutional layers the impact of SAFs in the inference accuracy is somewhat different. Each convolutional layer that is followed by a MaxPooling layer (i.e. Conv2, Conv4, Conv6) depicts a lower accuracy drop. This can be due to the fact that the max pooling layer can potential mask some of the faults, as it downsamples the inserted features by half. Therefore, in the case where a fault is present in the 2x2 kernel, it has a 50% chance of not propagating to the next layer, due to the pooling layer.



(a) Convolutional layers



(b) Fully connected (FC) layers.

Figure 4.5: SAF impact evaluation on VGG BNN architecture for the CIFAR-10 dataset.

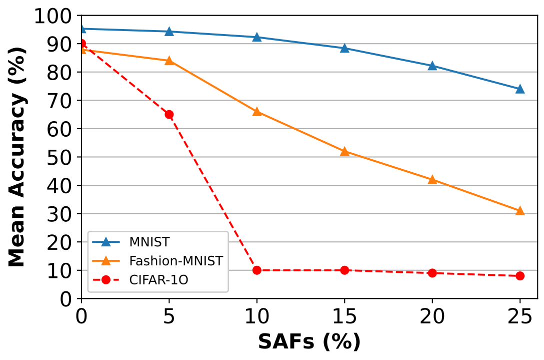


Figure 4.6: SAF impact evaluation on different datasets.

4.2.3 Evaluation over different datasets

In Figure 4.6 the impact of SAFs is evaluated for different datasets, namely MNIST, Fashion-MNIST and CIFAR-10. Overall, it is observed that injecting faults in a simpler BNN architecture for MNIST and in more complex BNNs for Fashion-MNIST and CIFAR-10 datasets, impacts the classification accuracy differently. For MNIST, the accuracy is reduced gradually with increasing fault rate, while it has a sharp accuracy reduction for Fashion-MNIST and CIFAR-10 datasets as their BNNs have more layers and neurons that are affected. For instance, for CIFAR-10, which is the most complex dataset to be classified among the test datasets, an accuracy reduction of 25% is noted for only 5% of SAFs. For higher fault rates the inference accuracy drops down to a very low percentage of about 10%, which is the random nominal accuracy for this dataset. Thus, we can argue that injecting faults in deeper and more complex BNNs impacts the classification accuracy greatly. This observation is also inline with the accuracy reduction of deep networks observed in Figure 4.4a.

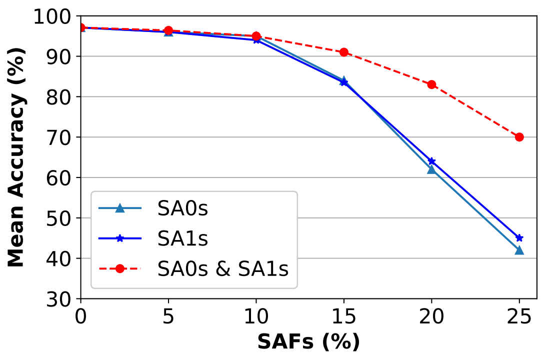


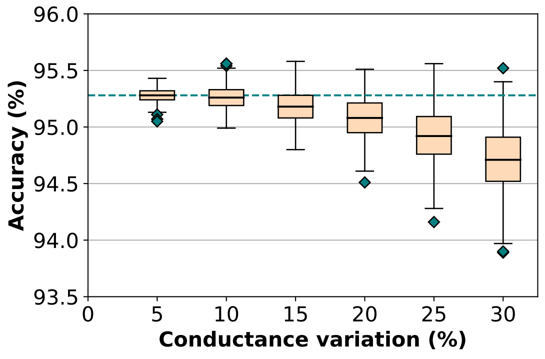
Figure 4.7: Different SAFs combinations on a 3-layer BNN for the MNIST dataset.

4.2.4 Evaluation of combinations of SA0s & SA1s

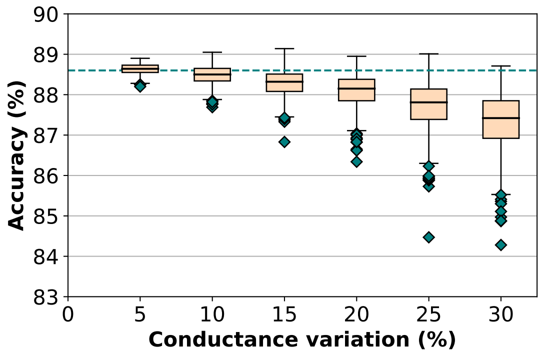
The independent impact of SA0 and SA1 faults is also studied separately as well as in a combined manner as shown in Figure 4.7. The figure shows the results from different distributions of SA0 and SA1 faults. From Figure 4.7, it is evident that when only SA0 or SA1 faults are injected, the accuracy degradation is almost uniform for both cases. However, when both SA0 and SA1 faults are applied simultaneously, which can happen in manufactured circuits, the accuracy reduction of the network is less than the reduction observed when SA0 and SA1 faults are injected independently. This is mainly driven due to the diminishing return effect, as both SA0 and SA1 can potentially cancel out (compensate) each other leading to less severe accuracy drop.

4.3 Conductance variation impact evaluation

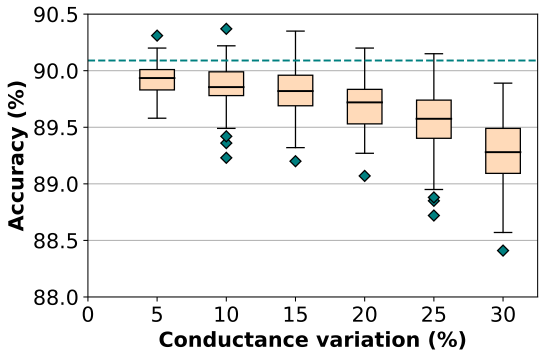
In this subsection the impact of conductance variation on the classification accuracy is evaluated. As discussed in the simulation framework (section 4.1.2), the conductance variation investigation focuses on device-to-device variations, with random samples drawn from a normal distribution. Figure 4.8 shows the impact of conductance variation (0-30%) on the inference accuracy of BNNs for the MNIST, Fashion-MNIST and CIFAR-10 datasets, figures 4.8a, 4.8b and 4.8c, respectively. In the figures the dotted line marks the fault-free inference accuracy in each dataset. In all cases, it is clear that conductance variation has rather minimal impact on the inference accuracy when compared to the insertion of SAFs. Notably, inserting 30% variation to the weights of the VGG BNN, results in a mean accuracy drop of only 0.8%. Similarly, for the MNIST and Fashion-MNIST dataset the accuracy decreases by approximately 1% when a maximum weight variation of 30% is considered. It is also worth to mention that inserting a higher percentage of variation to the weights causes an increasing deviation of the accuracy as well as more outliers. This is to be expected as high percentages of variation means that more randomness is added to the network, resulting in increasing variance from the mean case.



(a) 2-Layer BNN on the MNIST.



(b) 4-layer BNN on the Fashion-MNIST.



(c) VGG BNN on the CIFAR-10.

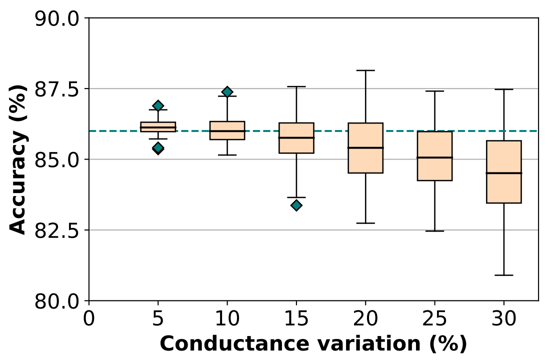
Figure 4.8: Impact of conductance variation on BNN's accuracy.

4.4 Combined impact of SAFs & conductance variation

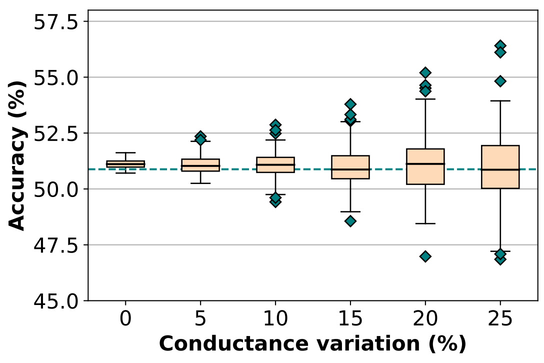
Apart from examining separately the impact of SAFs and variation, it is crucial to investigate the combined effect of SAFs and conductance variation on the inference accuracy of BNNs. For this purpose, the inference accuracy of the BNNs is evaluated in the presence of both SAFs and conductance variation for the MNIST and Fashion-MNIST. Experimental results for the CIFAR-10 dataset is not presented because the accuracy is heavily impacted by SAFs and therefore the impact of variations won't be visible. Figure 4.9a shows the accuracy reduction of the MNIST dataset on a 2-layer BNN with 15% SAFs while conductance variation is swept from 5% to 30% and figure 4.9a for a 4-layer BNN on the Fashion-MNIST dataset. It is clear that, the primary accuracy degradation is dominated by SAFs, while conductance variation has comparably minimum impact on the BNN's inference accuracy in both cases.

4.5 Summary and observations

In this chapter a thorough evaluation was conducted to assess the impact of RRAM faults on the inference accuracy of BNNs. First, we discussed about the architectural



(a) 2-Layer BNN on the MNIST.



(b) 4-Layer BNN on the Fashion-MNIST.

Figure 4.9: Impact of variation and 15% SAFs on BNN's inference accuracy.

implementation and the motivation of why using binary RRAM devices in combination with BNN architectures is a cost-effective hardware-friendly choice. Afterwards, the simulation framework was discussed: the selected BNN architectures and datasets were presented and the simulation of permanent (SAFs) as well as transient (drift and variation) faults were explained.

From the fault impact analysis it can be concluded that SAFs significantly degrade the accuracy of different BNN architectures (deep and shallow BNNs) for all datasets, while the accuracy drop due to conductance variation is rather minimal. As far as the impact of SAFs in different architectures is concerned, it is deduced that deeper BNNs are more vulnerable to high accuracy decline when subjected to SAFs than their shallow BNN counterparts. The same conclusion holds also for more complex datasets, such as the CIFAR-10. Therefore, for the MNIST dataset, for which we explored different architectures, we select the 2-layer BNN as the architecture over which we evaluate our fault-tolerance techniques later on, since from the fault impact analysis it is concluded that the 2-layer BNN is the most resilient architecture compared to the rest. For the Fashion-MNIST and CIFAR-10 datasets, we employ the 4-layer BNN and the VGG BNN, respectively.

What is more, the layer-wise investigation over both the MNIST and CIFAR-10 datasets concluded that faults injected in earlier fully-connected layers resulted in greater accuracy degradation. Thus, one could consider designing fault-tolerance techniques targeting only earlier stages of the fully-connected layers, as they play an important role in the accuracy drop.

Overall, the fault impact analysis is of great importance for this work as it exposes how RRAM faults influence the accuracy of BNNs and motivates the need for designing fault-tolerance techniques for implementing efficiently BNNs on RRAM-based crossbars.

Proposed Fault-Tolerance Techniques 5

In this chapter, the proposed fault-tolerance techniques are presented. The chapter begins with Section 5.1 by introducing the overall fault-tolerant design flow. After, the three fault-tolerance techniques are thoroughly discussed in Sections 5.2, 5.3 and 5.4 respectively.

5.1 Overall fault-tolerant design flow

Based on the analysis of the impact of RRAM defects on the inference accuracy of BNNs, given in Section 4.2, it is imperative to develop cost-effective fault-tolerance techniques in order to exploit the full potential of RRAM-based crossbar arrays for BNNs. For this purpose, a fault-tolerance framework is developed applying three different fault-tolerance techniques for RRAM-based crossbar arrays. The proposed fault-tolerant design flow is presented in Figure 5.1. Our goal is to mitigate both permanent faults (SAFs i.e. time-zero faults, and transient faults that become permanent) as well as transient faults which happen during run-time. As it can be seen from the figure, three main fault tolerance techniques are integrated, namely a fault-tolerant activation function, redundancy with a weight range adjustment and a retraining method. The framework first determines a fault-tolerant activation function, by evaluating different activation functions in the presence of injected hardware faults (SAF and/or conductance variation). The outcome of this stage (i.e., the fault-tolerant activation function) is used as a baseline activation function for the subsequent fault-tolerance techniques. These two techniques include a redundancy and weight range adjustment method for mitigating some of the present hardware faults, as well as a modified retraining technique. It should be noted that the presented techniques are orthogonal to each other, meaning that they can be applied together to guarantee higher accuracy restoration. Therefore, the main benefit of the proposed fault-tolerance framework is, that it enables to apply different levels of fault-tolerance techniques, allowing designers to make a trade-off between the accuracy improvement and their associated overhead.

5.2 A fault-tolerant activation function

An activation function is an essential part of a neuron, as it introduces non-linearity which ensures that a neural network can learn nonlinear behaviours [108, 109]. There are several activation functions in use, and among them three activation functions are most widely used: the *Sigmoid* function, the *hyperbolic tangent* or *Tanh* and the Rectified Linear Unit, or *ReLU*. Each of these activation functions have their own advantages and disadvantages from non-linearity, learning efficiency and implementation complexity, as described in section 2.1.2.3.

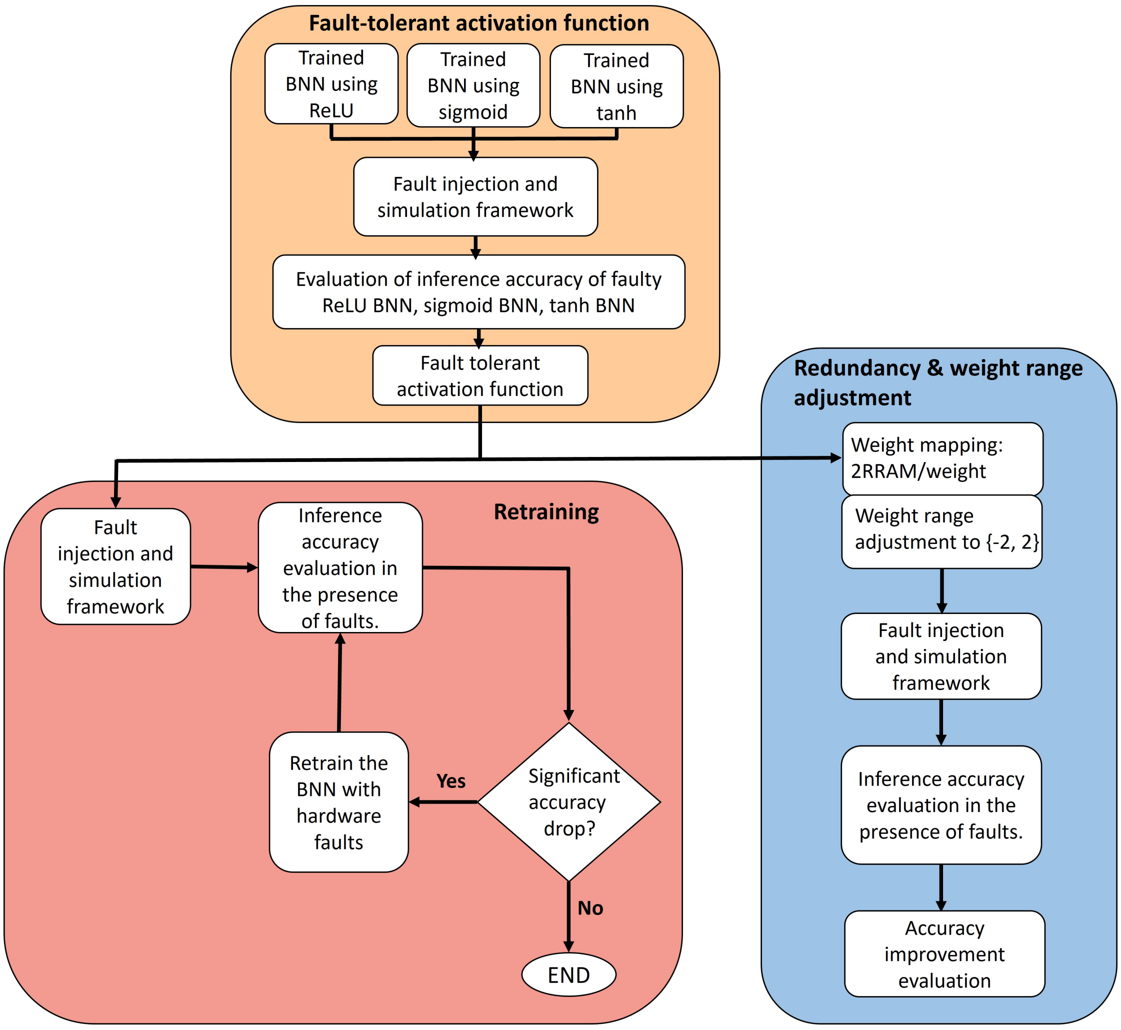

Fault-tolerant activation function
Trained BNN using ReLU
Trained BNN using sigmoid
Trained BNN using tanh
Fault injection and simulation framework
Evaluation of inference accuracy of faulty ReLU BNN, sigmoid BNN, tanh BNN
Fault tolerant activation function
Retraining
Fault injection and simulation framework
Inference accuracy evaluation in the presence of faults.
Significant accuracy drop?
Yes
Retrain the BNN with hardware faults
No
END
Redundancy & weight range adjustment
Weight mapping: 2RRAM/weight
Weight range adjustment to {-2, 2}
Fault injection and simulation framework
Inference accuracy evaluation in the presence of faults.
Accuracy improvement evaluation

Figure 5.1: Fault-tolerant design flow.

Moreover, the aforementioned activation functions also have different potential in tolerating faults in the underlying hardware. This fact is exploited to develop a design technique, that chooses an activation function by considering its fault-tolerance potential. Figure 5.1, presents the fault-tolerant activation function selection flow. Three pre-trained instances of BNNs using the *Sigmoid*, *Tanh* and *ReLU* functions are considered, and their fault tolerance capability is evaluated by mapping them to a faulty crossbar array using the evaluation framework presented in section 4.1. We make the assumption that each BNN is mapped on the same faulty crossbar array.

As mentioned in section 2.1.1 in table 2.1, both *Sigmoid* and *Tanh* functions produce non-sparse models as their neurons almost always are activated. That is because the produced output ranges are [0, 1] and [-1, 1], respectively, with the output not having a zero value or having a zero with a very low probability. Table 5.1 shows the number of active neurons, drawn from a simulation example of a 2-layer BNN trained with

Activation function	**Active neurons (%)**
Sigmoid	99.7
Tanh	100
ReLU	42.9

Table 5.1: The percentage of active neurons using a 2-layer BNN, when different activation functions are used.

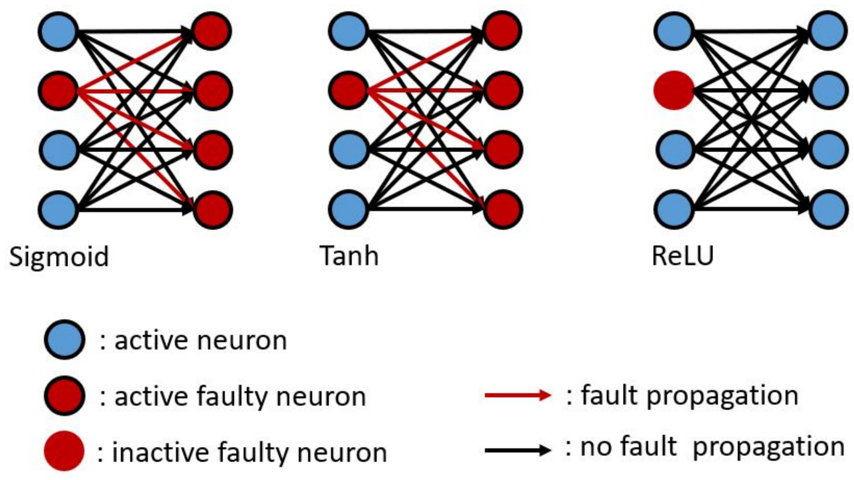
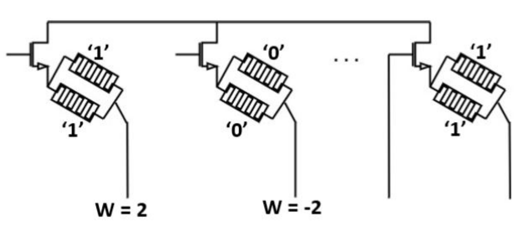


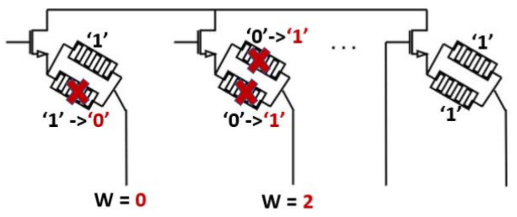
Figure 5.2: A visual example of fault propagation, when the *Sigmoid*, *Tanh* and *ReLU* activation functions are used.

the *Sigmoid*, *Tanh* and *ReLU* function on the MNIST dataset. The *active neurons* percentage refers to how many neurons are active on layer 1, on average over 10.000 test examples. In in the case of the BNN using the *Tanh* activation function all neurons are firing (100%), while for the *Sigmoid* BNN, 99.7% of neurons are active. For the *ReLU* BNN is is noted that around 42.9% of the neurons activate. Hence, for the *Tanh* and *Sigmoid* BNNs, from a fault-tolerance point of view, all (and almost all) faulty neurons fire with probability 1, fact that propagates faults to the consecutive layers. On the other hand, the *ReLU* function, has the advantage of adding sparsity to the network, resulting to a big percentage of the neurons (i.e 42.9%) to be inactive. This enables to mask some of the faults, thus preventing them from propagating to the next layers. As a result, it is expected that the BNN using the *ReLU* function to be more robust to faults than BNNs which use the *Sigmoid* and *Tanh* functions.

In figure 5.2 a simple visual example of three BNNs, when each of the activation functions is used, for showing how faults can propagate from one layer to the next. In this case, for simplicity, one fault is considered to be present in the three architectures. While the fault always propagates to the next layer when the *Sigmoid* and *Tanh* functions are used, as neurons always activate, the BNN using the *ReLU* manages to mask the fault, because the corresponding neuron remains inactive.



(a) Fault-free case.



(b) Faulty case: 1SAF/weight (W = 0), 2SAF/weight (W = 2).

Figure 5.3: Using 2RRAM devices per weight. The weights are adjusted based on the mapping method [11], to the set of values {-2, +2}.

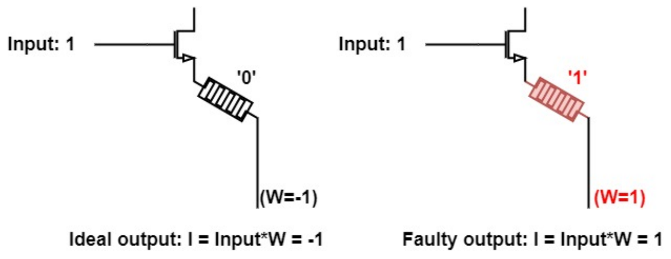

Input: 1
'0'
(W=-1)
Ideal output: I = Input*W = -1
Input: 1
'1'
(W=1)
Faulty output: I = Input*W = 1

Figure 5.4: The output of a neuron when 1RRAM per weight is used.

5.3 Redundancy & weight range adjustment

Redundancy is on of the common and widely used techniques for fault-tolerance. In this mitigation technique, we propose to use hardware redundancy for tolerating faults not by remapping the weights on secondary devices, but for increasing the range of the stored weight, thus minimizing the impact of a defective device, as shown in Figure 5.1. In the evaluation setup (section 4.1) it is stated that each RRAM device is used to represent the value of one weight. If a SAF is present, the mapped weight's value might flip from -1 to +1 and vise versa.

To realize the redundancy and weight adjustment based fault-tolerance method, the architecture of the crossbar is altered; from having 1T1R to 1T2R devices in parallel. As a result, each weight is mapped to two RRAM (2RRAM) devices. It is commented here that the devices should be connected in parallel in order for the proposed technique to work as expected. By connecting them in this way the logical 'OR' operation is essentially performed between the 2RRAM devices. If the RRAM devices were connected in series then the logical 'AND' operation would take place which won't mitigate any faults as explained below. Also, more devices could theoretically be connected in parallel, i.e. more than 2. However, the associated hardware overhead should be taken into account. Here towards proposing a cost-effective fault-tolerance technique we demonstrate the results only by using 2 RRAM devices per weight.

In order for the weight mapping on the RRAM crossbar to be valid, the weight's range is changed to {-2, +2}, as can be seen in figure 5.3. In the scenario that one SAF exists per weight, the weight's value of {-2, +2} will go to 0 and not {+2, -2} (figure

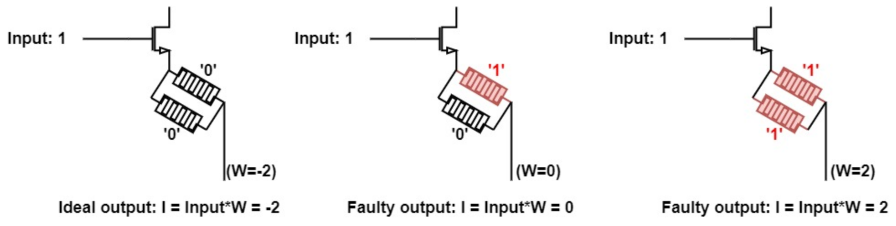

Input: 1
'0'
'0'
(W=-2)
Ideal output: I = Input*W = -2
Input: 1
'1'
'0'
(W=0)
Faulty output: I = Input*W = 0
Input: 1
'1'
'1'
(W=2)
Faulty output: I = Input*W = 2

Figure 5.5: The output of a neuron when 2RRAM per weight are used, for 1 fault and 2 faults per mapped weight.

5.3b). Therefore, we now have 3 possible values that the weight can hold {-2, 0, 2}. Compared to having the weights mapped to {-1, +1}, the extra state of 0 can potential make the faulty output sum of the neurons, closer to the fault-free result. A very simple example of the output of 1 neuron with only 1 weight, when 1RRAM device per weight is used is shown in figure 5.4. The output of a neuron when 2RRAM per weight are used, is presented in figure 5.5 respectively. Of course, if a high number of devices are faulty, the probability of having 2 SAFs per weight, is higher. In this case, the value of the weight would flip from $\pm$ 2 to $\mp$2, which is the same as the 1RRAM device case. This observation is confirmed by our experimental results presented in Figure 6.2.

5.4 Retraining

Retraining has been a widely adopted technique to compensate for the accuracy degradation of neural networks, due to various factors, such as hardware faults, transient errors, change in input features or operation conditions [110, 111]. For neural networks with online/on-chip learning features, retraining can restore the accuracy drop by altering the mapped weights during training. However, the main problem with online training is that it reduces the already limited lifetime of the RRAM devices due to multiple write operations [9]. On the other hand, for networks using offline training, the weights are optimized on software level and then have to be re-mapped to the RRAM crossbars. In this case, retraining can improve the network's accuracy significantly as shown in [32], while avoiding deterioration of the RRAM devices.

In this work we propose to combine an offline retraining method using BNNs with the fault-tolerance techniques presented as shown in Figure 5.1, to further enhance the inference accuracy of BNNs that are mapped to faulty RRAM crossbars. As presented in 5.1, first we evaluate the inference accuracy of the chosen BNN in the presence of RRAM faults. If the accuracy degradation is significant, we start the retraining procedure. Figure 5.6 presents in a more detailed way the steps of the retraining method. The first step of retraining is to extract the fault map of the RRAM-based crossbar arrays. In the present thesis, the fault map extraction is not addressed; i.e. it is assumed that the fault map is given (several methods have been proposed to extract the fault map, e.g. see [9, 32]). This map is used to initialize the corresponding weight matrices: the weights that are mapped to faulty devices are initialized to the corresponding fixed value. Afterwards, the training phase starts according to the BP algorithm. The output of each layer is computed in order to evaluate if the BNN is sufficiently trained or not. If the

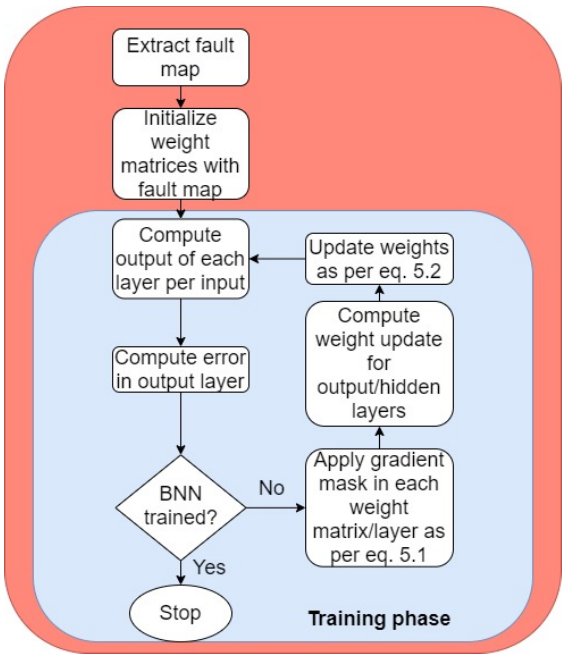
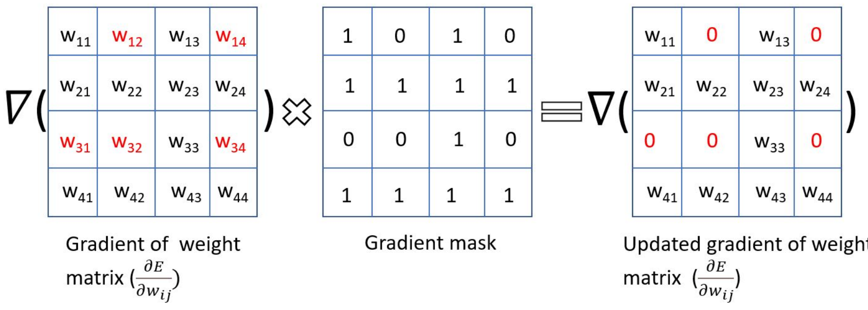

Extract fault map
Initialize weight matrices with fault map
Compute output of each layer per input
Compute error in output layer
BNN trained?
No
Yes
Stop
Apply gradient mask in each weight matrix/layer as per eq. 5.1
Compute weight update for output/hidden layers
Update weights as per eq. 5.2
Training phase

Figure 5.6: Retraining BNNs in the presence of RRAM faults.


Gradient of weight matrix ($\frac{\partial E}{\partial w_{ij}}$)
Gradient mask
Updated gradient of weight matrix ($\frac{\partial E}{\partial w_{ij}}$)

$$\nabla\left(\begin{array}{cccc} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{array}\right) \otimes \begin{array}{cccc} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{array} = \nabla\left(\begin{array}{cccc} w_{11} & 0 & w_{13} & 0 \\ w_{21} & w_{22} & w_{23} & w_{24} \\ 0 & 0 & w_{33} & 0 \\ w_{41} & w_{42} & w_{43} & w_{44} \end{array}\right)$$

Figure 5.7: The application of a gradient mask for freezing specific weights (w_{ij}) that are mapped on faulty RRAM devices .

BNN does not have high classification accuracy (meaning that it needs further training), the weights should be updated. For this purpose, before updating the weights, a mask (namely *gradient_mask*) is applied to the gradient of the weights ($\frac{\partial E}{\partial W_{ij}}$) to exclusively focus on retraining the weights mapped to non-faulty devices (equation 5.1), while faulty weights are 'frozen', as shown in equation 5.2. An example of using such a gradient mask is displayed in figure 5.7. By using this mask, it is made sure that the retraining phase updates only the fault-free weights (equation 5.2).

$$gradient\_mask = \begin{cases} 0, & \text{if weight} = SAF \\ 1, & \text{if weight} = fault - free \end{cases} \tag{5.1}$$

$$W_{ij}(t+1) = W_{ij}(t) - \eta \cdot \frac{\partial E}{\partial W_{ij}} \tag{5.2}$$

It is noted that in most cases, the usage of the gradient mask enables the network to converge faster as it has less parameters to update, which can be translated to faster retraining time. Moreover, for practical considerations, due to the fact that the retraining method can mitigate a large percentage of faults, we propose that it is implemented periodically, every time the accuracy is considered to have degraded significantly. That is, every time a severe accuracy degradation is detected, implying the presence of new SAFs, inference should stop in order to re-adjust the weights and remap them to the crossbar arrays.

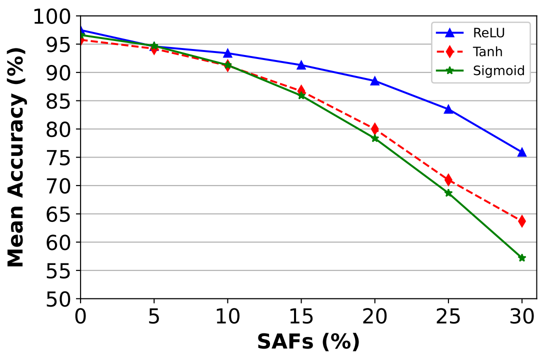
Results & Discussion 6

6.1 Accuracy improvement by the proposed fault-tolerance techniques

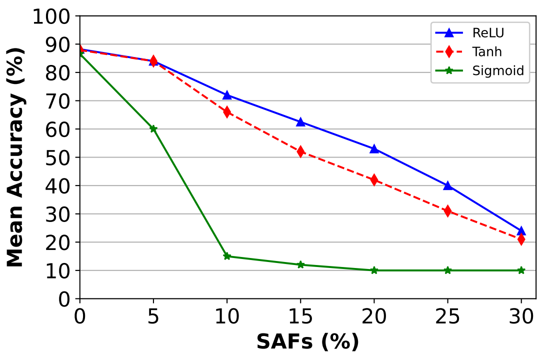
The effectiveness of the proposed techniques is evaluated using the architectural and simulation framework presented in chapter 4.1. This section presents the accuracy improvement results of the proposed techniques. First the results of the fault tolerant activation function are discussed, then the redundancy and retraining techniques are applied on top of it, to further restore the inference accuracy. It is noted that, for clarity of presentation, only the main simulation results are presented in the present section, to demonstrate the effectiveness of the proposed techniques. More simulation results are included in the Appendix section.

6.1.1 A fault-tolerant activation function

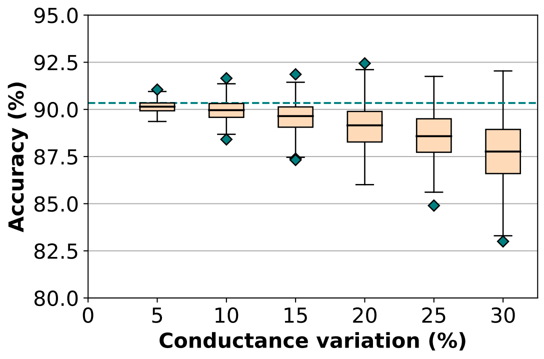
The fault tolerance capability of the three most commonly used activation functions, namely *ReLU*, *Tanh* and *Sigmoid* is evaluated and compared using the MNIST and Fashion-MNIST datasets as shown in Figure 6.1. As it can be seen from the plots in Figure 6.1 for the MNIST dataset, our insight on varying fault tolerance capabilities of different activation functions is validated from the simulation results. Figure 6.1 shows that both the *Tanh* and *Sigmoid* function exhibit similar accuracy reduction for the MNIST dataset across different fault rates, while the *ReLU* function leads to a smaller accuracy reduction. That is, using *ReLU* instead of *Tanh* or *Sigmoid* can yield up to 10% improvement without any additional fault tolerance techniques. Similarly, in the 4-Layer BNN trained for the Fashion-MNIST dataset shown in Figure 6.1, *ReLU* outperforms both *Tanh* and *Sigmoid*. In particular, for 10-25% SAF distributions the *ReLU* function has a 5-10% accuracy improvement over *Tanh*, while the *Sigmoid* function's performance is notably worse than both *Relu* and *Tanh*. Figure 6.1 demonstrates the performance of the *ReLU* function, in the presence of both SAF and conductance variation. Compared to figure 4.9a, we observe a certain accuracy improvement. However, it appears that this improvement is independent from the conductance varation's magnitude.The latter is to be expected, as inserting variation does not affect the accuracy significantly in the first place, as discussed in section 4.4. In general, the *ReLU* activation function gives better results in terms of accuracy, across a wide fault range and for varying network architectures and evaluation datasets. Nonetheless, it should be noted that switching the activation function in a more complex architecture, such as the VGG BNN for the CIFAR-10 dataset, did not yield better accuracy results. The reason for that is that the VGG BNN is severly affected by the fault injection as shown in figure 4.6, resulting in an accuracy drop down to around 10% for 10% of SAFs. Thus, different fault-tolerance



(a) MNIST dataset



(b) Fashion-MNIST dataset



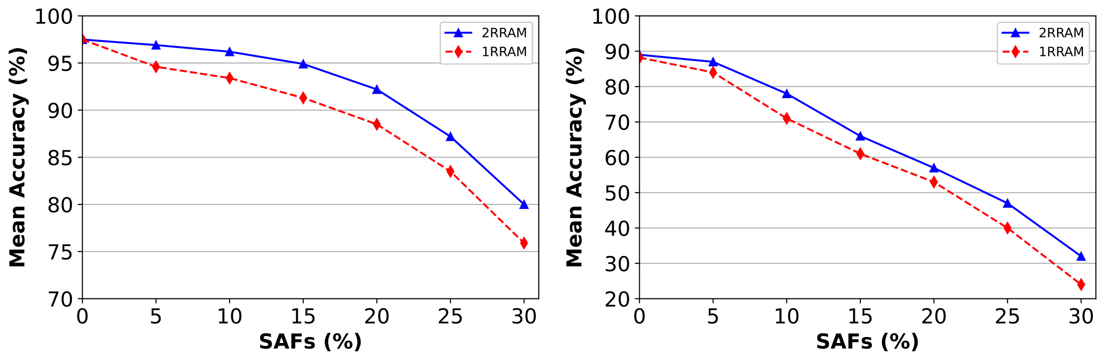
(c) Conductance variation + 15% SAF on 2-layer BNN ReLU for the MNIST dataset.

Figure 6.1: Evaluation of SAF and conductance variation impact on the inference accuracy using different activation functions.

techniques have to be utilized in order to efficiently restore the accuracy in such cases.

6.1.2 Redundancy & weight range adjustment

Having already chosen the fault-tolerant activation function, *ReLU* in this case, the proposed redundancy and weight range adjustment is evaluated over the MNIST and Fashion-MNIST datasets. The improved accuracy results of the redundancy technique for the MNIST and Fashion-MNIST, are presented in Figure 6.2. For the 2-layer BNN over the MNIST dataset (Figure 6.2(a)), redundancy led to a high accuracy (>95%), when up to 15% SAFs are inserted. For higher SAF rates (20-30%), the accuracy is reduced gradually, while maintaining a 5% improvement. Regarding the 4-layer BNN over the Fashion-MNIST (Figure 6.2(b)), the accuracy enhancement of the proposed redundancy and weight adjustment method varies depending on the rate of injected SAFs. Specifically, for SAF = 10% the redundancy approach improves the accuracy by ≈8%. This accuracy gain drops slightly to about 5% with more than 15% SAFs. As far as the VGG BNN is considered, the accuracy cannot be restored when this method is used. The reason is the same as to why the fault-tolerant activation function does not



(a) MNIST dataset (b) Fashion-MNIST dataset

Figure 6.2: Impact of SAFs using the redundancy and weight range adjustment technique.

perform well, as well.

6.1.3 Retraining

Table 6.1 presents the accuracy improvement results of the retraining method using the *ReLU* activation function for different datasets. From the table it can be observed that the retraining method is able to almost fully recover the accuracy back to the baseline fault-free accuracy (SAF 0% entry in the table), for all datasets and even for high fault-rates. In the case of the MNIST dataset for a 2-layer BNN, we can observe that even for a fault distribution as high as 30% the accuracy can be improved to 96.8% (from 71%), which is almost equal to the baseline accuracy (97.3%). For the Fashion-MNIST, the retraining method is able to recover the accuracy from a very low value of 22% up to 88.5%, which is almost ideal. Similar improvement is achieved for the CIFAR-10 dataset on VGG BNN, in spite of the complexity of the VGG BNN network.

It should also be mentioned that in some cases the accuracy achieved by the retraining method can even surpass the fault-free accuracy. The reason for that is that the retraining method forcefully alters the parameters of the network, since it fixes the values of the weights that correspond to faulty devices; and since back-propagation does not find a global minimum of the error function, implying that different parameters of the network may lead to different accuracy results, retraining can potentially lead to higher accuracy than the baseline one.

In order to evaluate what is the maximum fault percentage that the retraining method could restore, additional experiments were conducted with even higher random fault distributions, in particular up to 50%, for CIFAR-10 and Fashion-MNIST datasets, and up to 70% for the MNIST dataset. However, because such large distributions is not so realistic, they are not presented. It is worth mentioning, though, that the experiments surprisingly showed that fault distributions of up to 50% over the CIFAR-10 and 70% over the MNIST respectively could be compensated by the retraining method. This further motivates us to propose using the retraining method to mitigate not only time-

MNIST dataset			**Fashion-MNIST dataset**		**CIFAR dataset**	
SAF distribution (%)	Accuracy (%)		Accuracy (%)		Accuracy (%)	
	Baseline	Retrained	Baseline	Retrained	Baseline	Retrained
0	97.3	97.3	88.22	88.22	90.09	90.09
5	96	97.3	84	87.1	65	89.9
10	95.4	97	71	88	10	89.7
15	93	97.2	62	87.9	10	89.6
20	89	97.1	53	88	9	89
25	84	97.2	40	87.76	8	88.7
30	71	96.8	22	88.5	8	89

Table 6.1: Retraining with SAFs

zero faults but faults that are generated during runtime, thus being able to compensate for higher fault rates.

6.2 Overhead analysis & discussion

The effectiveness of the proposed fault-tolerant framework was evaluated is section 6 for different BNN architectures and datasets. The main objective was to develop a cost-effective fault-tolerant framework that would guarantee a reliable implementation of BNNs on RRAM-based crossbar arrays. As mentioned, the fault-tolerance framework starts by determining the activation function, *ReLU*, that is more robust to RRAM faults, to be used as a baseline function. A great advantage of this technique is that it is easy implementable in software level without adding any software or hardware overhead to the existing design. It is also evident that it yields significantly better results, notably an improvement of 10% and 8%, for the MNIST and Fashion-MNIST dataset respectively. Nonetheless, it should be acknowledged that there is a possibility that the designer wants to select a different activation function for their NN architecture; however, from the fault-tolerance point of view using the *ReLU* function is suggested.

As far as the redundancy and weight range adjustment technique is concerned, extra RRAM devices are utilized, not for remapping faulty weights as in other works, but for enlarging the represented weights' value set. This technique increases the accuracy by a maximum of 8%, without adding any software overhead. Another beneficial factor of this mitigation scheme is that it does not require any knowledge of the fault distribution, compared to the retraining techniques as mentioned above. However, modifications have to be made in the RRAM crossbar array in order to have 2 RRAM devices in parallel per weight, resulting in hardware overhead compared to the 1 RRAM crossbar implementation. Despite altering the weights from having values of {-1, +1} to {-2, +2}, it is noted that the BNN is not needed to be retrained for the new weights values. The only modification required is to divide each column's sum by 2, which can be easily realized by bit-shifting it to the left.

As discussed in section 6, the retraining technique restores the accuracy up to the ideal value in almost all test cases, even for a high percentage of faults. In contrast to similar retraining proposals, such as [8], the accuracy can be recovered even when faults cluster in a neighborhood in the RRAM crossbar. The main disadvantage of using retraining is that the fault distribution must be a-priori known in order to fixate the faulty weights during the back-propagation algorithm. However, certain techniques have been proposed that derive the fault-distribution: for example, a type March test [104], or an on-line fault detection phase [9]. Such methods could be combined with the proposed retraining method, even online. In fact, we propose to employ retraining in a periodic manner in order to be able to mitigate faults that occur during run-time as well. However, it is evident that the latter adds software overhead for the training algorithm as well as for detecting the fault distribution. In particular, the proposed retraining method (without considering the fault detection) adds a average latency of 390 seconds, 1200 seconds and 3300 seconds, for retraining the BNNs over the MNIST, Fashion-MNIST and CIFAR-10 datasets.

Overall, we can argue that our goal of developing a cost-effective fault-tolerance design is accomplished. Based on figure 5.1, the designer can select one and/or all fault-tolerant techniques depending on their accuracy requirements versus the hardware overhead or software trade-off. The proposed framework recovers the accuracy up to almost the ideal value for almost all test cases, while tackling one of the main limitations of other works; which is the requirement of complex mapping algorithms of the weights on the RRAM devices. Moreover, due to the fact that BNNs are employed instead of costly floating-point DNNs, we reduce the number of RRAM devices required for mapping the NN right from the start, sparing additional hardware resources.

6.3 Comparison with state-of-the-art techniques

In Table 6.2, a comparison to other state-of-the art fault tolerance techniques [32], [31], [33] is presented. The comparison is conducted using a 2-layer NN for the MNIST dataset, while having 20% SAFs. In the table, the term “recovered accuracy” is the ratio of the restored accuracy divided by the baseline accuracy, and the redundancy column (R) refers to the extra RRAM devices for representing each one of the weights.

In short, the proposed technique in [32], uses a modified retraining method to incorporate the RRAM faults during the training phase, similar to the technique proposed in the present work. In [32], the authors consider the significance of the NN’s weights with respect to the impact they have on the classification accuracy and propose to remap significant weights that are originally mapped on faulty RRAM devices to redundant ones, to further improve the classification accuracy. While [32] recovers the classification accuracy up to 95.1%, our work restores the inference accuracy up to 99.8%.

Authors in [31] propose a mapping algorithm (MAO), in order to map the weights on the RRAM based crossbar, that minimizes the mapping error. MAO recovers the accuracy up to 43%, without any hardware overhead. However, to achieve better results a redundancy scheme is utilized (R=2), resulting in recovering the accuracy up to 96%. This redundancy scheme utilizes either whole secondary columns or entire crossbar arrays.

In [33], three different matrix transformations are used to tackle the impact of SAFs,

Similar Works	**Recovered Accuracy (%)**	**Retraining**	**Redundancy (R)**
[32]	95.1	Yes	1
[31]	43	No	1
	96		2
[33]	30.1	No	1
	97.6		2
Our proposal	99.8	Yes	1
	94.6	No	2

Table 6.2: Comparison with related fault tolerance techniques, using a 2-Layer NN for the MNIST dataset, with 20% SAFs.

combined with extra RRAM devices (R=2) for the weight mapping. First, a row flipping and a permutation transformation is used, followed by a value range transformation of the weights. All three techniques target to maximize the NNs accuracy as well as to minimize the mapping error between the weights and the corresponding RRAM devices, in the presence of SAFs. When no redundancy is used, the recovered accuracy remains low (30.1%), whereas for R=2 the accuracy is restored up to 97.6%. Both methods [33], [31] add software overhead for implementing the mapping algorithm and the matrix transformations, while imposing also hardware overhead (R=2), for a better accuracy result.

Overall, our proposed fault-tolerant framework outperforms the related works, restoring the accuracy up to 99.8%, with a comparatively less overhead, when the retraining technique and the fault-tolerant activation function is applied (R=1 row in the table). Apart from restoring significantly the classification accuracy, our approach does not use extra costly mappings of the weights to the RRAM based crossbar arrays, by instead using binary weights mapped on binary RRAM devices, in contrast to weights which are of higher precision assumed in the related works [31, 32, 33] (e.g. 6/8 bit or floating point). If the redundancy and weight adjustment technique is considered along with the fault-tolerant function, but not the retraining technique (R=2 row in the table), we observe that the accuracy can be restored to the satisfactory value of 94.6%. It should be noted that our redundancy technique compared to the works of [33], [31], does not require any additional transformations for mapping and approximating the weights with the corresponding RRAM devices, resulting into less hardware and software expenses.

Conclusion & Future Outlook 7

In this Chapter we conclude the work of this thesis. First, we summarize the main points of this work in Section 7.1. Afterwards, in Section 7.2 the contributions of our proposed framework are discussed, followed by a discussion concerning the limitations of the current work as well as future work in Section 7.3.

7.1 Overview

In this thesis our main goal is to develop cost-effective fault-tolerance techniques in order to map accurately NNs on RRAM based crossbar arrays. Having already conducted a thorough literature review in Chapter 3, regarding the state-of-the-art fault tolerance techniques, we identified two main limitations: a) a significant software overhead concerning finding the optimal mapping between the weights and the corresponding RRAM devices, and b) significant hardware cost originating from the need for redundant devices to achieve higher rates of fault-tolerance.

Towards developing low-cost solutions, we chose a specific class of NNs, namely BNNs, which are lighter versions of traditional NNs as explained in Chapter 2.1.2.3. Due to the fact that BNN's weights can obtain either of only two values $\pm$ 1, binary RRAM devices were also selected, for mapping each weight using 1 RRAM device. This assumption eliminates the software overhead stemming from the complex mapping algorithms as in [8], [31]. Furthermore, binary weights mapped on RRAM devices (1 RRAM device per weight), require 8 or 6 times less hardware area to be realized, compared to higher resolution weights of 8 or 6 bits respectively.

With the architectural implementation in place, different BNN architectures are evaluated using three different datasets: the MNIST, Fashion-MNIST and CIFAR-10. In the RRAM fault impact analysis we consider SAFs originating from the manufacturing stage and transient faults that become permanent during runtime. Apart from SAFs, variation in the RRAM devices nominal values is also considered. By taking into account both permanent and transient issues we can simulate a large amount of RRAM defects with the SAF model as well as with conductance variation respectively. The evaluation results showed that deeper BNN architectures exhibit a higher inference accuracy degradation due to RRAM faults, while shallower networks proved to be more fault-tolerant. Moreover, from the fault impact analysis we observed that SAFs have a significant effect on the accuracy of all BNNs, while variability problems do not impact the accuracy of BNNs notably.

Motivated by the fault impact analysis, fault-tolerance solutions are developed in Chapter 5 in order to recover the inference accuracy of BNNs. In the direction of finding low-cost solutions, the fault tolerance framework proposes three mitigation techniques: 1) a fault tolerant activation function, 2) redundancy and weight range adjustment and,

3) a retraining technique. The fault-tolerant activation function technique evaluates different activation functions (*Tanh*, *Sigmoid* and *ReLU*) in the presence of SAFs and variations. As discussed in Chapter 5, the *ReLU* function produces sparse BNN architectures, which is beneficial from a fault-tolerance point of view. Therefore, the *ReLU* function is selected as the baseline activation function. Afterwards, the redundancy and weight range adjustment scheme is presented. This method suggests using 2 RRAM devices per weight instead of 1 RRAM device, to enlarge the set of possible weight's values. The latter alleviates, to some extent, the negative impact of RRAM faults in the inference accuracy of BNNs. Lastly, a modified retraining technique is proposed for enhancing the fault-tolerance capabilities of the mapped BNN. In a nutshell, this method incorporates the RRAM faults during the training phase of the BNN. For this purpose, the fault distribution map of the RRAM based crossbar array should be known a-priori.

The results of the fault-tolerant framework are presented in Chapter 6 along with a comparison with the state-of-the-art fault-tolerance techniques. Starting with the fault-tolerant activation function, *ReLU* shows great results for the MNIST and Fashion-MNIST datasets. In particular for the MNIST dataset, an accuracy improvement of 10% is observed compared to the *Tanh* and *Sigmoid*, when 15% SAFs are injected. For the Fashion-MNIST, in the presence of 15% SAFs, the *ReLU* function improves the accuracy by 10% and 50% respectively compared to the *Tanh* and *Sigmoid* functions. The second fault tolerance method, for SAF $= 10\%$, improves the accuracy by $\approx 5\%$ and $\approx 8\%$ for the MNIST and Fashion-MNIST datasets. Unfortunately, when it comes to the VGG BNN for the CIFAR-10 dataset, the accuracy cannot be restored remarkably with the first 2 fault-tolerance techniques. Therefore, a retraining method is proposed to enhance further the fault-tolerance capabilities of the proposed framework. By using this retraining method, all BNNs can recover their inference accuracy, in the presence of very high fault rates (up to 30%), close to their ideal accuracy value. A great advantage of the proposed fault-tolerant framework is that the user is free to choose which fault-tolerance methods best suit to their case, based on the trade-off between accuracy improvement and hardware overhead. Overall, we can argue that our fault-tolerant framework achieves its objectives; to build cost-effective solutions that restore the inference accuracy of various BNN architectures.

7.2 Contributions

The contributions of this work can be summarized as follows:

- An evaluation setup was developed for studying the impact of RRAM faults on different BNN architectures and datasets. The evaluation setup is divided in two parts:

  1. We developed a low-cost architectural implementation, using binary RRAM devices and BNNs to be executed on the RRAM-based crossbar arrays, thus tackling the need for complex mapping algorithms, which is present in related works [8, 31]. Moreover, our assumption utilizes 8 times less hardware area compared to works using 8bit accuracy weights, where each weight requires 8 RRAM devices. On the other hand, in the case where multilevel RRAM

devices are considered, e.g. 8 bit devices [112], binary RRAM devices are still preferable because they are easier to program and less prone to faults [105].

2. The simulation framework includes the BNN architectures and datasets used for evaluating the impact of the RRAM faults. While other related works use one NN architecture/dataset for evaluation [8], [32], [33], in this work 3 different datasets, namely MNIST, Fashion-MNIST and CIFAR-10, are used along with an FP2BIN, an n-layer BNN (n=2,3,4) and a VGG BNN architecture. A variety of SAF and variation rates were also investigated, concluding that SAFs is the main factor affecting the accuracy of the networks. Moreover, from the layerwise investigation we deduced that faults inserted in earlier layers of the BNNs degrade the accuracy significantly.

Overall, the fault impact analysis provided valuable insights regarding how SAFs and conductance variation affect the inference accuracy of a variety of BNN architectures.

- A fault tolerance framework was developed motivated by the conducted fault impact analysis as well as by the limitations found in similar state-of-the-art proposals. To be precise, our fault tolerance framework adheres to our main goal, which was to develop cost-effective solutions which are able to restore the inference accuracy of the BNNs sufficiently. In particular:

  1. The first fault-tolerance technique, the fault-tolerant activation function, achieves to mitigate some of the RRAM faults, in a simple and inexpensive manner in terms of hardware and software overhead. Changing only the activation function of the network from $Tanh$ to $ReLU$, improves the inference accuracy by 10% for the MNIST and 15% for the Fashion-MNIST dataset.
  2. The weight and range adjustment technique is based on using 2 RRAM devices per weight, not for remapping faulty devices as previous works suggest, but for extending the set of values that a single weight can hold. This method improves the inference accuracy by a maximum of around 5% for the MNIST and 8% for the Fashion-MNIST dataset. Although the accuracy is increased with the expense of doubling the number of RRAM devices, no further transformation or assumption is needed as in other related works using redundant devices.
  3. The third fault-tolerance technique, namely the retraining method, has already been explored by related works [32], [8] in a similar manner. However, these works fail to restore the accuracy when a high number of faults cluster near a neighborhood in the weight matrix or the fault rate is very high ($\geq$ 20%). In the present work, the retraining method recovers the inference accuracy back to the ideal value for all 3 datasets and BNN architectures, even for high fault rates of 30%. Furthermore, compared to other works, we use retraining for mitigating both SAFs generated from the manufacturing stage and during runtime.

- A comparison with state-of-the-art works was also conducted, with our proposed fault-tolerant framework yeilding better results, with lower associated overhead.
- To the best of our knowledge, the related literature on combining binary RRAM devices and BNN architectures for better fault tolerance solutions is very scarce [38], and the present work further contributes on that front.

7.3 Discussion and future work

Although this work has reached to some interesting results and conclusions, further improvements should be made in order to overcome current limitations and drawbacks.

To begin with, almost all simulation results were evaluated using a high level analysis with Python. This high level approach is dominant in most related works in the literature, mainly because of the abstraction and hence the simplicity it provides. However, it is advised that more low/circuit level simulations are conducted, in order to have more realistic and accurate results. Performing circuit simulations would also incorporate the influence of the periphery (e.g. DACs, ADCs etc) when calculating the matrix-vector multiplication results.

The faults that were considered, SAFs and variation, can model a relatively big variety of RRAM defects. In the case of conductance variation, the model used with the normal distribution, is abstracting the variability phenomenon occurring in RRAM devices to a great extent. The latter is beneficial from a simplicity point if view, but lacks on accuracy. More specifically, different models can be used for simulating device-to-device and cycle-to-cycle variability problems as in [11], [105]. Using specific models might capture more accurately the behavior of RRAM devices. In that respect, one could explore more RRAM non-idealities, e.g. non-linear write operations, in greater detail as well as non-idealities originating from the RRAM crossbar array and the corresponding periphery. Moreover, even though the evaluation made by the present work is somewhat thorough compared to other works, by considering many different datasets and BNN architectures, there is still room for considering more datasets such as the ImageNet, and different NN architectures.

Although the proposed fault-tolerance framework has indeed shown promising results, there is still room for improvement. The fault-tolerant activation function is a simple and efficient way to mitigate some of the RRAM faults in relatively simple multi-layer BNN architectures. When it comes to more complex architectures such as the VGG which involves convolutional as well as pooling layers, the result is not satisfying. This is to be expected as even a small amount of SAFs (5%) degrades the accuracy significantly by around 30%, whereas moving to higher faults rates renders the VGG enable of classifying any examples, with an accuracy of $\approx$ 10%.

The redundancy and weight range adjustment technique that we developed requires further circuit simulation in order to evaluate more accurately the hardware overhead and implementation details. We proposed that the 2 RRAM devices will be connected in parallel in each intersection of the crossbar array. As a result, the drain of the transistor where the RRAM device is usually connected to has to be prolonged. This implementation detail should be further investigated. In this regard, one could also explore the possibility of using redundant columns of RRAM devices, connected in such a way that the weighted sum of the two columns is added. Having redundant columns instead of RRAM devices connected in parallel, might solve the previously mentioned implementation issue, but it will yield hardware overhead due to the extra periphery and transistors. Furthermore, the use of more than 2 RRAM devices could also be investigated, in order to possibly enhance the fault-tolerance capabilities of the method. For instance, if the hardware overhead of using higher number of redundant devices (≥ 2) is acceptable, the fault impact on more complex networks such as the VGG BNN could also be addressed via redundancy.

The main drawback of the retraining scheme is that the fault distribution of the RRAM based crossbar arrays must be known beforehand. The main question that arises is: *how to obtain an accurate fault map?* Unfortunately, most of the related retraining methods don't specify how the fault map is obtained. Authors in [32] acquire the fault distribution once by utilizing chip testing, while [9] introduces an online fault detection scheme. In our case, we assume that the fault map can be obtained in a periodic manner, to achieve the highest possible fault-tolerance. Thus, further research should be conducted on how the fault detection phase should be implemented during runtime. We could approach this problem similarly to [9], by developing a fault detection phase based on a *quiescent-voltage comparison* method. Alternatively, a different approach could be designing a Build-In-Self-Test (BIST) inside the crossbar array as in [113].

Bibliography

[1] J. Chen and X. Ran, “Deep learning with edge computing: A review.” *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[2] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[3] H. Sutter *et al.*, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobb’s journal*, vol. 30, no. 3, pp. 202–210, 2005.

[4] R. Nagyfi. The differences between artificial and biological neural networks. [Online]. Available: https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7

[5] S. Saha, “A comprehensive guide to convolutional neural networks,” 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[6] F. Sultana, A. Sufian, and P. Dutta, “Evolution of image segmentation using deep convolutional neural network: A survey,” *CoRR*, vol. abs/2001.04074, 2020. [Online]. Available: https://arxiv.org/abs/2001.04074

[7] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, “Metal–oxide rram,” *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[8] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, “Accelerator-friendly neural-network training: Learning variations and defects in rram crossbar,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 19–24.

[9] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang, “Fault-tolerant training with on-line fault detection for rram-based neural computing systems,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[10] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[11] X. Sun and S. Yu, “Impact of non-ideal characteristics of resistive synaptic devices on implementing convolutional neural networks,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 570–579, 2019.

[12] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, “Evolving deep neural networks,” in *Artificial intelligence in the age of neural networks and brain computing*. Elsevier, 2019, pp. 293–312.

[13] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations,” in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[14] M. A. Lebdeh, U. Reinsalu, H. A. Du Nguyen, S. Wong, and S. Hamdioui, “Memristive device based circuits for computation-in-memory architectures,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[15] H. A. Du Nguyen, J. Yu, L. Xie, M. Taouil, S. Hamdioui, and D. Fey, “Memristive devices for computing: Beyond cmos and beyond von neumann,” in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2017, pp. 1–10.

[16] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, “Towards a serverless platform for edge {AI},” in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[17] J. Chen and X. Ran, “Deep learning with edge computing: A review.” *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[18] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, “Memristor for computing: Myth or reality?” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 722–731.

[19] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, p. 107281, 2020.

[20] J. Bethge, H. Yang, M. Bornstein, and C. Meinel, “Binarydensenet: developing an architecture for binary neural networks,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[21] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor, “A classification of memory-centric computing,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 2, pp. 1–26, 2020.

[22] S. Hamdioui *et al.*, “Memristor based computation-in-memory architecture for data-intensive applications,” in *DATE*, 2015.

[23] I. Giannopoulos *et al.*, “In-memory database query,” *Advanced Intelligent Systems*, 2020.

[24] A. Rahman, J. Lee, and K. Choi, “Efficient fpga acceleration of convolutional neural networks using logical-3d compute array,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1393–1398.

[25] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, “Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision,” in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018, pp. 218–220.

[26] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 10, pp. 1864–1878, 2014.

[27] G. Charan *et al.*, "Accurate inference with inaccurate RRAM devices: A joint algorithm-design solution," *Journal on Exploratory Solid-State Computational Devices and Circuits*, 2020.

[28] E. I. Vatajelu, P. Prinetto, M. Taouil, and S. Hamdioui, "Challenges and solutions in emerging memory testing," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 3, pp. 493–506, 2017.

[29] W. Zhang *et al.*, "Design guidelines of rram based neural-processing-unit: A joint device-circuit-algorithm analysis," in *DAC*, 2019.

[30] W. Shim, Y. Luo, J.-S. Seo, and S. Yu, "Investigation of read disturb and bipolar read scheme on multilevel rram-based deep learning inference engine," *IEEE Transactions on Electron Devices*, vol. 67, no. 6, pp. 2318–2323, 2020.

[31] W. Huangfu, L. Xia, M. Cheng, X. Yin, T. Tang, B. Li, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang, "Computation-oriented fault-tolerance schemes for rram computing systems," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 794–799.

[32] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[33] B. Zhang, N. Uysal, D. Fan, and R. Ewetz, "Handling stuck-at-faults in memristor crossbar arrays using matrix transformations," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 438–443.

[34] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on rram," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 782–787.

[35] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Enabling ai at the edge with xnor-networks," *Communications of the ACM*, vol. 63, no. 12, pp. 83–90, 2020.

[36] ——, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[37] X. Sun, S. Yin, X. Peng, R. Liu, J.-s. Seo, and S. Yu, "Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1423–1428.

[38] A. Chaudhuri, B. Yan, Y. Chen, and K. Chakrabarty, "Hardware fault tolerance for binary rram crossbars," in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[39] C. J. B. Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[40] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[41] L. Xia, W. Huangfu, T. Tang, X. Yin, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang, "Stuck-at fault tolerance in rram computing systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 102–115, 2017.

[42] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[43] J. Nolte, *The human brain*. Mosby/Elsevier,, 1993.

[44] M. Seunggu Han. All you need to know about neurons. [Online]. Available: https://www.medicalnewstoday.com/articles/320289

[45] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[46] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[47] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.

[48] D. Liu, "A practical guide to relu," 2017. [Online]. Available: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

[49] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[50] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[51] G. E. Hinton, T. J. Sejnowski *et al.*, *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[52] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[53] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[54] X. Ying, “An overview of overfitting and its solutions,” in *Journal of Physics: Conference Series*, vol. 1168, no. 2. IOP Publishing, 2019, p. 022022.

[55] Y. Yao, L. Rosasco, and A. Caponnetto, “On early stopping in gradient descent learning,” *Constructive Approximation*, vol. 26, no. 2, pp. 289–315, 2007.

[56] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, “Speech recognition using deep neural networks: A systematic review,” *IEEE Access*, vol. 7, pp. 19 143–19 165, 2019.

[57] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, “Object detection with deep learning: A review,” *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[58] W. Rawat and Z. Wang, “Deep convolutional neural networks for image classification: A comprehensive review,” *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[59] H. Gholamalinezhad and H. Khosravi, “Pooling methods in deep neural networks, a review,” 2020.

[60] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[61] S. Srinivas and R. V. Babu, “Data-free parameter pruning for deep neural networks,” *arXiv preprint arXiv:1507.06149*, 2015.

[62] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[63] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, p. 107281, 2020.

[64] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.

[65] A. Bulat and G. Tzimiropoulos, “Xnor-net++: Improved binary neural networks,” *arXiv preprint arXiv:1909.13863*, 2019.

[66] X. Lin, C. Zhao, and W. Pan, “Towards accurate binary convolutional neural network,” *arXiv preprint arXiv:1711.11294*, 2017.

[67] R. Ding, T.-W. Chin, Z. Liu, and D. Marculescu, “Regularizing activation distribution for training binarized deep networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 408–11 417.

[68] S. Darabi, M. Belbahri, M. Courbariaux, and V. P. Nia, “Bnn+: Improved binary network training,” 2018.

[69] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 365–382.

[70] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[71] Z. Wang, J. Lu, C. Tao, J. Zhou, and Q. Tian, "Learning channel-wise interactions for binary convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 568–577.

[72] J. Faraone, N. Fraser, M. Blott, and P. H. Leong, "Syq: Learning symmetric quantization for efficient deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4300–4309.

[73] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua, "Quantization networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7308–7316.

[74] P. Wang, Q. Hu, Y. Zhang, C. Zhang, Y. Liu, and J. Cheng, "Two-step quantization for low-bit neural networks," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2018, pp. 4376–4384.

[75] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[76] X. Zhang and Y. LeCun, "Text understanding from scratch," *arXiv preprint arXiv:1502.01710*, 2015.

[77] A. Gupta, A. Anpalagan, L. Guan, and A. S. Khwaja, "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues," *Array*, p. 100057, 2021.

[78] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, pp. 24–29, 2019.

[79] H. Mohsen, E.-S. A. El-Dahshan, E.-S. M. El-Horbaty, and A.-B. M. Salem, "Classification using deep learning neural networks for brain tumors," *Future Computing and Informatics Journal*, vol. 3, no. 1, pp. 68–71, 2018.

[80] J. Wen, E. Thibeau-Sutre, M. Diaz-Melo, J. Samper-González, A. Routier, S. Bottani, D. Dormont, S. Durrleman, N. Burgos, O. Colliot *et al.*, "Convolutional neural networks for classification of alzheimer's disease: Overview and reproducible evaluation," *Medical image analysis*, vol. 63, p. 101694, 2020.

[81] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition.* Ieee, 2009, pp. 248–255.

[82] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[83] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 77–84.

[84] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[85] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[86] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[87] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[88] R. Waser, "Redox-based resistive switching memories," *Journal of nanoscience and nanotechnology*, vol. 12, no. 10, pp. 7628–7640, 2012.

[89] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Communications of the ACM*, vol. 53, no. 7, pp. 99–106, 2010.

[90] G. Fuchs, N. Emley, I. Krivorotov, P. Braganca, E. Ryan, S. Kiselev, J. Sankey, D. Ralph, R. Buhrman, and J. Katine, "Spin-transfer effects in nanoscale magnetic tunnel junctions," *Applied Physics Letters*, vol. 85, no. 7, pp. 1205–1207, 2004.

[91] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* IEEE, 2005, pp. 459–462.

[92] X. Sun, X. Peng, P.-Y. Chen, R. Liu, J.-s. Seo, and S. Yu, "Fully parallel rram synaptic array for implementing binary neural network with (+ 1,- 1) weights and (+ 1, 0) neurons," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 574–579.

[93] S. Hamdioui *et al.*, "Memristor for computing: Myth or reality?" in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 722–731.

[94] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[95] A. Chen, "A review of emerging non-volatile memory (nvm) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016.

[96] J. Borghetti *et al.*, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[97] L. Xie *et al.*, "Scouting logic: A novel memristor-based logic design for resistive computing," in *ISVLSI*, 2017.

[98] J. Yu *et al.*, "Enhanced scouting logic: a robust memristive logic design scheme," in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 2019, pp. 1–6.

[99] R. Liu, H. Wu, Y. Pang, H. Qian, and S. Yu, "A highly reliable and tamper-resistant rram puf: Design and experimental validation," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 13–18.

[100] I. Chakraborty, M. F. Ali, D. E. Kim, A. Ankit, and K. Roy, "Geniex: A generalized approach to emulating non-ideality in memristive xbars using neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[101] J.-H. Lee *et al.*, "Exploring cycle-to-cycle and device-to-device variation tolerance in MLC storage-based neural network training," *Transactions on Electron Devices*, 2019.

[102] W.-Q. Pan, J. Chen, R. Kuang, Y. Li, Y.-H. He, G.-R. Feng, N. Duan, T.-C. Chang, and X.-S. Miao, "Strategies to improve the accuracy of memristor-based convolutional neural networks," *IEEE Transactions on Electron Devices*, vol. 67, no. 3, pp. 895–901, 2020.

[103] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, "Rxnn: A framework for evaluating deep neural networks on resistive crossbars," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[104] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2014.

[105] W. Shim, Y. Luo, J.-s. Seo, and S. Yu, "Impact of read disturb on multilevel rram based inference engine: Experiments and model prediction," in *2020 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2020, pp. 1–5.

[106] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[107] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[108] D. Misra, "Mish: A self regularized non-monotonic neural activation function," *arXiv preprint arXiv:1908.08681*, vol. 4, 2019.

[109] S. Sharma, "Activation functions in neural networks," *towards data science*, vol. 6, 2017.

[110] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[111] J. Deng, Y. Rang, Z. Du, Y. Wang, H. Li, O. Temam, P. Ienne, D. Novo, X. Li, Y. Chen *et al.*, "Retraining-based timing error mitigation for hardware neural networks," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 593–596.

[112] F. Zahoor, T. Z. A. Zulkifli, and F. A. Khanday, "Resistive random access memory (rram): an overview of materials, switching mechanism, performance, multilevel cell (mlc) storage, modeling, and applications," *Nanoscale research letters*, vol. 15, no. 1, pp. 1–26, 2020.

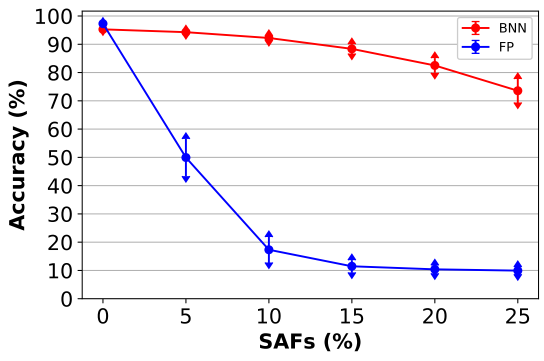
[113] X. Cui, M. Zhang, Q. Lin, X. Cui, and A. Pang, "Design and test of the in-array build-in self-test scheme for the embedded rram array," *IEEE Journal of the Electron Devices Society*, vol. 7, pp. 1007–1012, 2019.

[114] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.

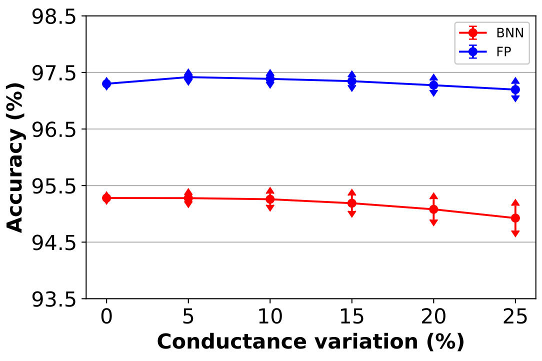
A Comparison of BNN and Floating-Point (FP) architectures in the presence of RRAM faults.

In this appendix a comparison between BNN and FP architetcures is conducted, to further motivate the use of BNNs from a fault-tolerance point of view. Computation-In-Memory (CIM) using RRAM crossbar arrays is a promising solution to realize energy-efficient neuromorphic hardware. In this regard, BNNs, where the weights are single binary values, are inherently suitable for CIM-based acceleration. Along with that, we can argue that BNNs are inherently more fault-tolerant compared to traditional FP NN architectures. A simulation example using a 2-layer NN for the MNIST dataset is presented, in order to evaluate the fault-tolerance in both the binary and floating-point network. In the case of binary weights, we consider a weight to be permanent faulty if its value is stuck-at a $\pm$ 1 (LRS/HRS) as described in section 4.2. Considering that a SA0 and a SA1 fault occurs randomly with the same probability, there is a 50% chance that a faulty weight has actually the correct value, which is an important advantage from a fault-tolerance point of view. On the other hand, for a FP NN architecture, a permanent faulty weight would mean that the weight's value is stuck either at the maximum or minimum value of the corresponding weight matrix, as considered also in [8].

Figure A.1 shows the simulation accuracy result when the SAF rate spans from 0 to 25 % for the BNN and the FP NN architecture. Even for a high fault rate of 25%, the BNN depicts significant fault-tolerant capabilities compared to the corresponding FP NN architecture. In particular, the accuracy drops to 71%, while in the traditional



(a) SAF evaluation.



(b) Conductance variation evaluation.

Figure A.1: Impact of fault injection on the accuracy of a BNN and FP NN architecture.

NN the accuracy degradation is much larger, reaching 10%. Apart from evaluating the impact of SAFs, variation in the weights' value is also examined in both architectures, as shown in figure A.1b. Compared to inserting SAFs, variation does not have such a significant impact on the NN's accuracy in both architectures. Overall, we can claim that using a BNN architecture on binary RRAM-based crossbars is preferable considering the fault-tolerance aspect compared to the traditional NN architectures.

Additional experiments for Chapter 4. B

Although, the results presented in the main text of this thesis are not considering any pre-processing methods, here some extra experimental results showing the impact of SAFs on the inference accuracy, when pre-processing methods are used, are presented. The pre-processing texhnique used is Principal Component Analysis (PCA) In a nutshell, PCA is a widely used method for reducing the input dimensionality of datasets, while minimizing information loss. This is done by creating new uncorrelated variables [114]. In the case of BNN mapped on RRAM-based crossbar arrays, using PCA is worth to be investigated. This is because PCA reduces the input vector, thus reducing the size of the corresponding RRAM crossbar array. A reduced size in the RRAM crossbar, implies that less hardware is used for realizing the BBN architecture. Moreover, the fact that PCA transforms the input data in more re-presentable ones, reduces the training time of the network, as it becomes lighter. Table B.1 presents the accuracy of a 2-layer BNN over the MNIST dataset with different input feature sizes. Reducing the input size to 149 yields a accuracy of 93.2 % close to the original accuracy. Therefore, the 2-layer BNN with 149 input features is evaluated when SAFs are injected (figure B.1).

#Features	**Accuracy(%)**
784 (original)	95.28
236	89.47
149	**93.2**
64	93
98	92
23	86

Table B.1: Comparing the classification accuracy of a 2-layer BNN over the MNIST dataset for different features sizes.

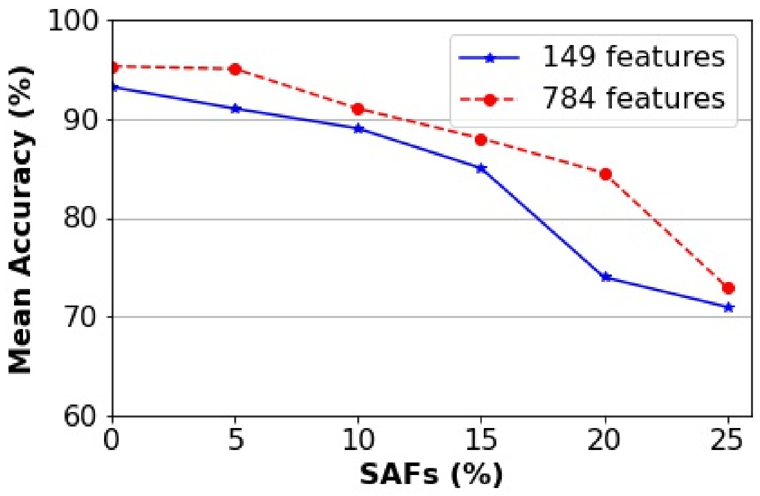


Figure B.1: SAF impact evaluation of a 2-layer BNN consisting of 149 inoput features. The 149 input feature BNN shows similar accuracy degradation behavior as the original 2-layer BNN with 784 input features.

Brief paper C

In this Appendix a brief paper is attached, which summarizes the present thesis and will be submitted for publication in the near future.

RRAM Crossbar-Based Fault-Tolerant Binary Neural Networks (BNNs)

***Abstract*—Computation-In Memory (CIM) using RRAM crossbar array is a promising solution to realize energy-efficient neuromorphic hardware. In this regard, Binary Neural Networks (BNNs), where the weights are single binary values, are inherently suitable for CIM-based acceleration. However, RRAM faults, due to variability and endurance, restrict the applicability of CIM. To address this issue, we propose different fault tolerance techniques to mitigate the impact of RRAM faults and defects on the accuracy of CIM-based BNN hardware. These techniques include a fault tolerant activation function, redundancy and weight range adjustment and a retraining technique to regain the BNN's accuracy reduction, due to RRAM faults. Evaluation results using MNIST, Fashion-MNIST and CIFAR-10 datasets demonstrate that the proposed techniques can improve the inference accuracy in the presence of RRAM defects by up to 20%, 40% and 80%, respectively.**

***Index Terms*—Computation-In-Memory (CIM), fault tolerance, RRAM**

I. INTRODUCTION

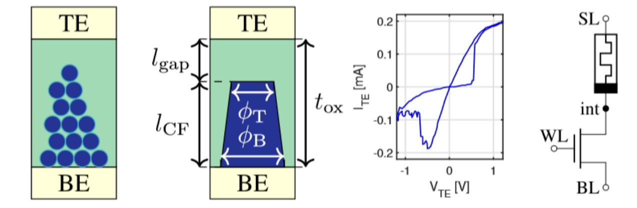
Computation-In-Memory (CIM) using emerging Resistive Random Access memory (RRAM), which integrates computation and storage in the same physical location, has emerged as a promising solution to tackle the DNN challenges and enable their deployment on resource constrained platforms [1]. However, RRAM devices have various defects and faults such as conductance drift, read disturb and device non-idealities which limit their widespread applicability [2]. Hence, addressing these issues is of paramount importance to realize reliable DNN operations on RRAM-based CIM hardware.

RRAM defects can be broadly classified as *permanent defects*, defects at *t=0* due to fabrication challenge or *transient defects* defects at *t>0* due to device variability or runtime issues. During operational time, however, the *transient defects* can become *permanent defects*.In order to mitigate the impact of these defects, several software and hardware-based fault tolerance approaches have been proposed [3], [4], [5], [6], [7], [8], [9]. Some of the software-based solutions focus on finding an optimal mapping between the weights and the RRAM devices [3], [4], while other researchers focus on retraining techniques to partially regain the accuracy reduction [5], [6]. The failure map of the RRAM devices is obtained by utilizing a quiescent voltage comparison method [6], while in [5] the location of the defective devices is acquired by chip testing.Similarly, the hardware-based solutions utilize redundancy schemes to improve fault tolerance [7], [8], [9]. In [8], a re-configurable redundancy scheme is proposed, using redundant columns or crossbar arrays, for tolerating SAFs. In [9], first a March-type test [10] is used to obtain the fault map of the crossbar, followed by a modified 4T1R architecture for tolerating random SAF distributions. However, these solutions have various limitations, such as significant software or hardware overhead, and need for complex mapping algorithms. Moreover, these techniques ignore the impact of conductance variation. Therefore, there is a clear need for a comprehensive, cost-effective and efficient fault tolerance technique in order to deploy DNNs in RRAM-based CIM crossbar array.

In this paper we propose cost-effective and efficient fault tolerance techniques addressing the impact of RRAM defect-induced Stuck-at fault (SAF) and conductance variation on the inference accuracy of DNNs with binary weights commonly referred as Binary Neural Networks (BNNs). The paper first motivates the need for fault tolerance techniques on BNNs mapped to CIM crossbar arrays by investigating the impact of RRAM faults and conductance variation on the inference accuracy of BNNs. Then, three fault tolerance techniques are proposed to mitigate the impact of RRAM faults and conductance variation on the BNN's inference accuracy. The first mitigation technique investigates the role of different activation functions on suppressing the impact of RRAM faults on the BNN's accuracy, and demonstrates how a fault-tolerance aware activation function can help to restore the accuracy. The second technique explores the potential of redundancy combined with a weight range adjustment for fault-tolerant BNN design, followed by a retraining technique to regain the RRAM fault induced accuracy reduction. The main contributions of the paper are summarized as follows:

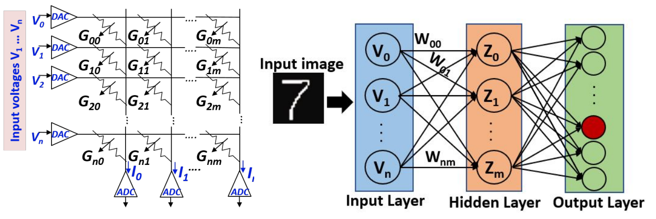
- Analyse the impact of SAFs on inference accuracy.
- Analyse the impact of conductance variation on inference accuracy.
- Evaluate the sensitivity of different BNN architectures to SAFs and conductance variation.
- Demonstrate the role of activation functions for fault tolerance.
- Develop and demonstrate low-cost and complementary fault tolerance techniques.

The reminder of the paper is organized as follows: Section II presents the background of RRAM devices, basics of CIM-based neural network implementation, and state-of-the-art works on fault tolerance techniques. Section III demonstrates the impact of RRAM faults on the BNN's inference accuracy. Section IV presents the proposed mitigation schemes followed by the discussion on the achieved accuracy improvement results in Section V. Finally the paper is concluded in SectionVI.



(a) Ox. vacancies (b) Tech. parameters (c) I-V curve (d) 1T-1R cell

Fig. 1: Structure of a RRAM device and its working principles [19].



(a) RRAM crossbar array (b) Two layer NN

Fig. 2: RRAM crossbar array for CIM-based NN (a) crossbar structure (b) Demonstration of two layer NN mapping to RRAM crossbar.

II. Background

A. RRAM device basics

Resistive Random Access Memory (RRAM) devices have recently gained widespread attention due to their non-volatility, high integration density and their ability to overcome memory bandwidth issues by executing operations within the memory [11]. These properties make RRAM attractive for various applications ranging from non-volatile memory [12], logic based around computation in memory [13], [14], [15], [16], [17], and neuromorphic computing [18]. A RRAM device as shown in Figure 1 is fabricated by sandwiching a metallic oxide (commonly HfO$_x$, or TiO$_x$) between the two regions, i.e., the doped region (top electrode) and the undoped region (bottom electrode). The size of the CF determines the resistance state of the device. When a sufficiently high positive voltage (higher than the set threshold voltage, V$_{set}$) is applied, a CF as shown Figure 1(a)) is formed and the device will have a low resistance state. On the contrary, when a negative voltage (lower than reset threshold voltage, V$_{reset}$) is applied, the CF is broken which leads to high resistance state (Figure 1(b)) Figures 1(c) and (d) show the I-V curve of RRAM and the most prevalent one transistor one resistor (1T1R) structure of RRAM devices.

B. Computation-In-Memory (CIM) based NNs

In addition to their storage functionality, RRAM devices can be used to build a computation unit by structuring them in a crossbar array. Figure 2(a) shows a crossbar structure with N wordlines (input voltages) and M bitlines, in which the wordlines and bitlines are connected through a RRAM device at their intersection. The crossbar shown in Figure 2(a) can be used to perform Matrix-Vector Multiplication (MVM) operation by applying a voltage vector $V{=}V_j$ (where $j{\in}\{1,N\}$) to the RRAM crossbar matrix of conductance values $G{=}G_{ij}$ (where $i{\in}\{1,N\}$, $j{\in}\{1,M\}$). At any instance, each column performs a vector-vector multiplication (VVM) or a MAC operation, with the output current vector I, in which each element is $I_i{=}\Sigma V_j \cdot G_{ij}$. Note that all M MAC operations are performed with O(1) time complexity.

This O(1) MVM computation potential of CIM provides fast and efficient computing power for different neural network applications such as Binary Neural Networks (BNNs) kernels. Figure 2(b) shows a simplified fully-connected 2-Layer BNN consisting of input , hidden and output layers. The neurons in every layer calculate the weighted sum of the input vector, and then pass it through an activation function by adding bias value as shown in Equation (1). From Equation (1) we can observe that the weighted sum operation can be efficiently accelerated using RRAM crossbar array shown in Figure 2(a).

$$O_i = f(\sum W_{ij} \cdot V_i + b_i) \qquad (1)$$

where O$_i$ is the output of the neuron i, f is the activation function, $\sum W_{ij} \cdot V_i$ is the weighted sum of the input values and their respective weights and b_i is the bias value.

C. Defects in RRAM devices

Despite their significant advantages as described in previous sections, RRAM devices suffer from various reliability and variability problems, hindering their large scale manufacturing. RRAM defects can be divided in two categories: permanent and transient. In the case of a permanent defect, the RRAM device is stuck in either a high or low resistance state. The RRAM device's value can not be tuned anymore with a write operation. Root causes include overforming of the device [20] and open defects, such as a permanent open switch [2]. Apart from permanent defects originated from the manufacturing stage, transient defects can also become permanent. For instance, multiple read or write operations can disturb the state of the RRAM device making it in the course of time, get stuck at a low or high resistance state []. On the other hand, transient defects are dynamic, meaning that the state of the RRAM device is not fixed in a particular resistance value, but can be tuned and changed from time to time during run-time. Intermittent problems are linked to the inherent properties of RRAM devices, attributed to the stochastic nature of the oxygen vacancies and ions processes []. Therefore, process variation is a significant root cause of intermittent defects []. Variation makes the device's resistance value differ from its ideal value and can be cycle-to-cycle (C2C) or device-to-device (D2D). D2D variation occurs when different RRAM devices show different resistance characteristics under identical programming conditions [21], while C2C variation happen when a single RRAM device shows different resistance characteristics from time to time [22]. Moreover, the accumulated effect of large numbers of read/write operations can lead to significant change (drift) of the resistance state of RRAM devices [23], which can be considered an transient defect. At last, the non-linear and asymmetric I-V characteristics of resistive memories (shown in Figure 1) causes the RRAM device to deviate from its ideal value. In particular, variation in a read voltage can lead to different effective resistance ratios, causing functional errors [24].

III. RRAM Faults Impact Analysis

In this section we will analyze the impact of RRAM defects on the inference accuracy of BNNs. In this analysis, we will consider the impact of Stuck-at-fault (SAF) and conductance variation only, as they can aggregate the impact other RRAM defects. First the architectural and simulation setup is discussed. Next, the impact of SAFs and variation on the BNNs' accuracy is discussed.

A. Evaluation setup

1) Architecture implementation: Two aspects are important to implement BNNs on RRAM-based CIM crossbar arrays: a) the structure of the RRAM crossbar array and b) the BNN architecture. In this work we consider a binary 1T1R bit-cell design, in which the cell can hold 2 states; HRS(0)/ LRS(1). Binary RRAMs are easier to realize and program than multilevel RRAM, making them also less prone to manufacturing and operational faults [25]. In this work a specific category of Neural Networks, BNNs (with weights ± 1) is chosen to study the impact of RRAM faults and demonstrate the effectiveness of the proposed fault-tolerance techniques. Since BNNs use binary weights, they can easily be mapped to RRAM-based crossbar array as, each weight can be stored as a conductance value in one RRAM device.

$$\underbrace{\begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix}^\top}_{\text{weights}} \cdot \underbrace{\begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix}}_{inputs} = 2 \cdot \underbrace{\begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_n \end{bmatrix}^\top}_{conductances} \cdot \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} - \underbrace{\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}^\top}_{col\_ones} \cdot \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} \tag{2}$$

$$W_i = \begin{cases} -1, & \text{if } G_i = 0 \text{ (HRS)} \\ 1, & \text{if } G_i = 1 \text{ (LRS)} \end{cases} \tag{3}$$

It is evident though that a one-to-one mapping of the BNN's weights (W_i) to the conductance values (G_i) is not feasible as the value of -1 has to be stored in an RRAM cell. For this purpose, we use the mapping scheme shown in equation (2) [26], which is implemented by adding an extra column of RRAM cells (*col_ones*) in the crossbar array as shown in Figure 3. The *col_ones* holds unit conductance values ($G_{unit_i} = 1$). We take the weighted output sum of the crossbar array multiplied by 2, (i.e. the term $2 \cdot G_i \cdot V_i$ in equation 2) and subtract from it the output of the extra column. This scheme allows us to map both -1 and +1 weight values to the RRAM conductance states. The selected mapping technique has the advantage of reduced hardware overhead, compared to other methods, such as in [3], where 2 crossbar arrays are utilized for holding positive and negative values for the weights.

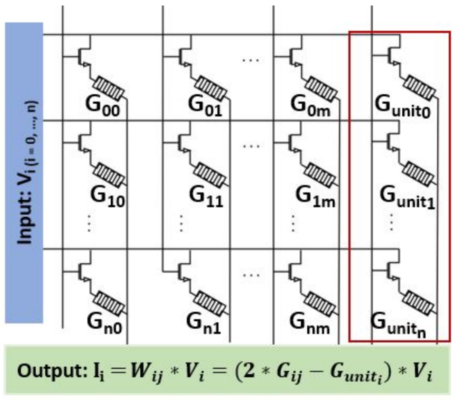


Fig. 3: Implementation of selected mapping [26] on RRAM based crossbar array. The red frame shows the added extra column (G_{unit_i}) for realizing weights = -1.

2) Simulation setup: The simulation setup of this work is presented in Table I. In this evaluation different network architectures such as n-layer BNN (n=2, 3, 4), Floating Point 2 Binary network (FP2BIN) and VGG BNN are used along with MNIST, Fashion-MNIST and CIFAR-10 datasets. As shown in the table, different rates of SAF and conductance variation are evaluated. We assume that SAFs are inserted randomly in the weights' array, as -1s (SA0s) and 1s (SA1s) based on the aforementioned mapping. The number of faults injected in the network is equal to the percentage of faults multiplied by the total number of weights. In this investigation only device-to-device variations are considered, as they have severe impact than cycle to cycle variations. Device-to-device variations are simulated by replacing the ideal weights with a random value selected from a normal distribution as shown below.

$$weight \sim \mathcal{N}(\mu, \sigma^2),$$

where:

$$\mu = \begin{cases} -1, & \text{if weight} = -1 \\ 1, & \text{if weight} = 1 \end{cases}$$

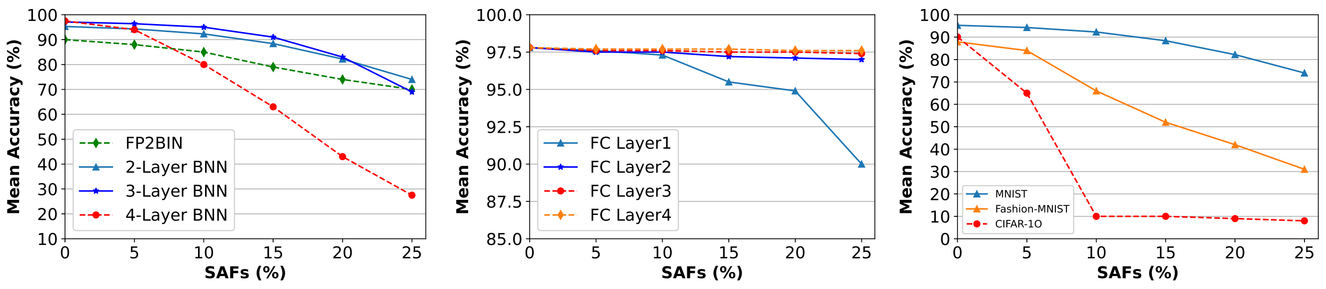
Moreover, a layer-wise fault analysis is conducted in order to determine how the injecting faults in different layers of the network affect the classification accuracy.

B. Stuck-at-fault (SAF) impact evaluation

The impact of SAF (both SA-0 and SA-1) on the inference accuracy is evaluated for different network architectures using

TABLE I: Evaluation Setup: BNNs, datasets and faults investigated

Architecture	# Layers	Dataset	Baseline Accuracy (%)
FP2BIN	2	MNIST	90
n-Layer BNN (n = 2, 3, 4)	2-4		95.28, 97.1, 97.8
4-Layer BNN	4	Fashion-MNIST	88.6
VGG BNN	9	CIFAR-10	90.09



(a) Deep vs shallow network (b) Layer-wise SAF analysis (c) Different datasets

Fig. 4: SAF impact evaluation on different architecture of BNN and varying datasets

the MNIST, Fashion-MNIST and CIFAR-10 datasets as shown in Figure 4. Figure 4(a) shows the impact of SAF on shallow and deeper networks, for four different architectures, of BNNs using the MNIST dataset. From the figure we can observe that the SAF-induced accuracy reduction increases with increase in the network depth, i.e. the more hidden layers, the higher the SAF impact on the network inference accuracy. This is mainly due to the fact that the layers are fully connected, which enable faults to affect more neurons which ultimately can severely harm the accuracy of deeper BNNs. Therefore, we can safely conclude that the 2-Layer BNN is a relatively resilient architecture to SAF than the deeper BNN variants.

The impact of SAFs injected in a layer-wise manner is also investigated in this work as shown in Figure 4(b). The figure shows the accuracy reduction of a 4-layer BNN, by applying all the faults in a single layer at a time. From Figure 4(b) we can observe that faults injected in earlier layers led to higher accuracy reduction, than faults in the later stages. This could be potentially due to two factors: 1) faults in earlier stages can affect more features (e.g., in the first layer). 2) the impact of the faults can propagate to the later stages leading to higher accuracy reduction, while the faults in later stages (e.g., layer 4) affect few neurons. In Figure 4(c) the impact of the SAFs is evaluated for different datasets, namely MNIST, Fashion-MNIST and CIFAR-10. Overall, we can observe that injecting faults in simple BNNs for MNIST and complex BNNs for Fashion-MNIST and CIFAR-10 datasets, impacts the classification accuracy differently. For MNIST, the accuracy is reduced gradually with increase in fault rate, while it has a sharp accuracy reduction for Fashion-MNIST and CIFAR-10 datasets as their BNNs have more layers and neurons. This observation is inline with the accuracy reduction of deep networks observed in Figure 4(a).

The independent impact of SA0 and SA1 faults is also studied separately as shown in Figure 6. The figure shows the results from different distributions of SA0 and SA1 faults. From Figure 6, it is evident that when only SA0 or SA1 faults are injected the accuracy degradation is almost uniform for both cases. However, when both SA0 and SA1 faults are applied simultaneously, which can happen in manufactured circuits, the accuracy reduction of the network is less than the reduction observed when SA0 and SA1 faults are injected independently. This is mainly driven due to the diminishing return effect, as both SA0 and SA1 can potentially cancel out (compensate) each other leading to minimum accuracy reduction.

C. *Conductance variation impact evaluation*

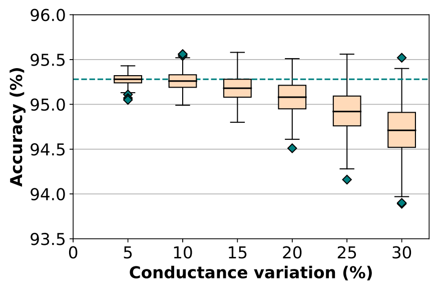
In this subsection the conductance variation impact on the classification accuracy is evaluated. As discussed in the setup subsection, the conductance variation investigation focuses on device-to-device variations using a normal distribution. Figure 5, shows the impact of conductance variation (0-30%) on the inference accuracy of BNNs for MNIST, Fashion-MNIST and CIFAR-10 datasets, figures 5(a), 5(b) and 5(c), respectively. In all cases, it is clear that conductance variation has rather minimal impact on the inference accuracy when compared to SAFs. Moreover, it is crucial to investigate the combined effect of SAFs and conductance variation on the inference accuracy of BNNs. For this purpose, the inference accuracy of the BNNs is evaluated in the presence of both SAFs and conductance variations. Figure 7 shows the accuracy reduction of MNIST dataset on a 2-layer BNN with 15% SAFs while conductance variation is swept from 5 to 30%. It is clear that, the accuracy degradation is dominated by SAFs, while conductance variation has comparably minimum impact on the BNN's inference accuracy.

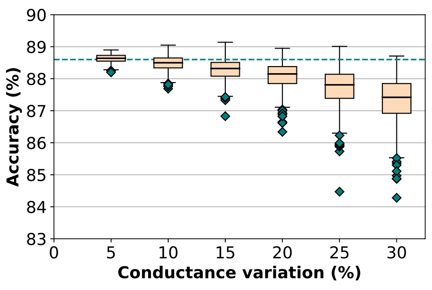
Overall, it can be concluded that SAFs significantly degrade the accuracy of different BNN architectures (deep and shallow BNNs) for all datasets, while the accuracy drop due to conductance variations is rather minimum. Regarding the impact of SAFs in different architectures, we can deduce that deeper BNNs are more vulnerable to high accuracy reduction when subjected to SAF than their shallow BNN counterparts. Therefore, from the fault impact analysis it can be observed that a 2-layer BNN is relatively resilient to hardware faults. Hence, it is used as a baseline to evaluate the fault tolerance techniques proposed in this paper, which will be discussed in the subsequent section.

IV. Proposed Fault-Tolerant Techniques

A. *Overall fault tolerance flow*

Based on the analysis of the impact of RRAM defects on the inference accuracy of BNNs, given in Section III, it is


(a) 2-Layer BNN on MNIST dataset


(b) 4-Layer BNN on Fashion-MNIST

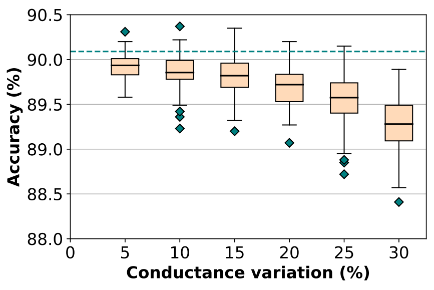

(c) VGG BNN on CIFAR-10

Fig. 5: Impact of conductance variation on BNN's accuracy.

imperative to develop fault tolerance techniques in order to harness the full potential of RRAM-based crossbar arrays for BNNs. For this purpose, we develop a fault-tolerance framework that applies different fault tolerance techniques for RRAM-based crossbar arrays. The proposed fault tolerance flow is presented in Figure 8. As it can be seen from the figure, three main fault tolerance techniques are integrated, namely fault-tolerant activation functions, redundancy with weight range adjustment and a retraining method. The framework first determines a fault-tolerant activation function, by evaluating different activation functions in the presence of injected hardware faults (SAF and/or conductance variation). The outcome of this stage (i.e., the fault-tolerant activation function) will be used as a baseline activation function for the other fault-tolerance techniques. It should be noted that the presented fault tolerance techniques are orthogonal to each other, meaning that they can be applied together to guarantee higher accuracy. Therefore, the main benefit of the proposed fault tolerance framework is, that it enables to apply different levels of fault tolerance techniques, allowing designers to make a trade-off between accuracy improvement and their associated overhead.

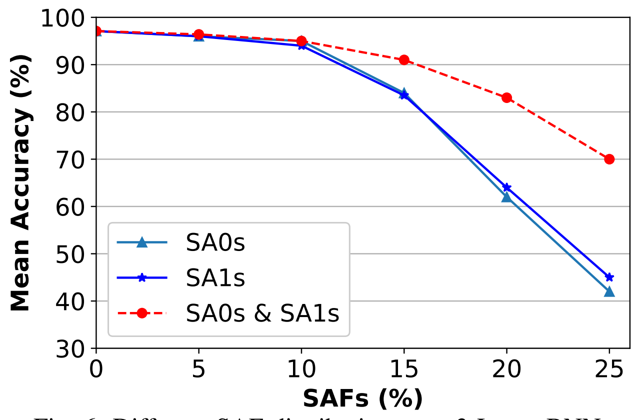


Fig. 6: Different SAF distribution on a 3-Layer BNN.

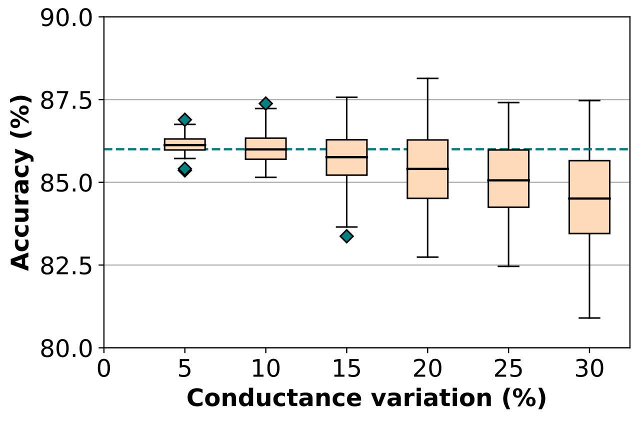


Fig. 7: Impact of variations and 15% SAFs on 2-Layer BNN.

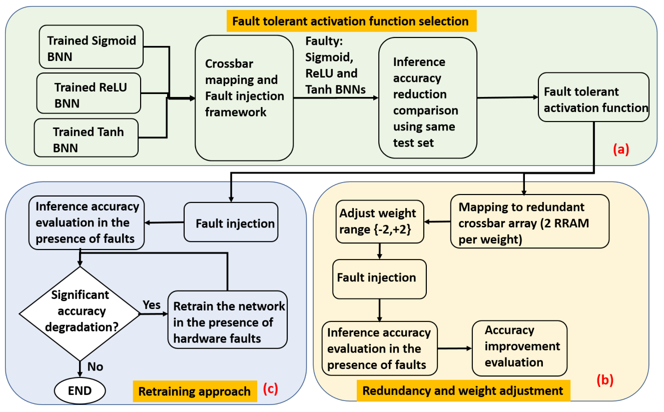
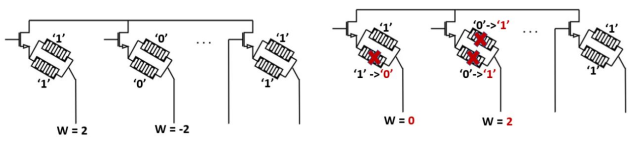

Fault tolerant activation function selection
Trained Sigmoid BNN
Trained ReLU BNN
Trained Tanh BNN
Crossbar mapping and Fault injection framework
Faulty: Sigmoid, ReLU and Tanh BNNs
Inference accuracy reduction comparison using same test set
Fault tolerant activation function
(a)
Inference accuracy evaluation in the presence of faults
Fault injection
Significant accuracy degradation?
Yes
Retrain the network in the presence of hardware faults
No
END
Retraining approach
(c)
Adjust weight range {-2,+2}
Mapping to redundant crossbar array (2 RRAM per weight)
Fault injection
Inference accuracy evaluation in the presence of faults
Accuracy improvement evaluation
Redundancy and weight adjustment
(b)

Fig. 8: The proposed fault tolerant flow implementation of a trained BNN.

B. *Fault-tolerant activation function*

An activation function is an essential part of a neuron, that defines the output of the neuron, given set of inputs, by taking the weighted sum of the input values. In addition to that, an activation function introduces non-linearity which ensures that a neural network can learn nonlinear behaviours [27], [28]. There are several activation functions in use, and among them three activation functions are most widely used: the $Sigmoid$ function, the hyperbolic tangent or $Tanh$ and the Rectified Linear Unit, or $ReLU$. Each of these activation functions have their own advantages and disadvantages from non-linearity, learning efficiency and implementation complexity. Moreover, the aforementioned activation functions also have different potential in tolerating faults in the underlying hardware.



(a) Fault-free case.

(b) Faulty case: 1SAF/weight (W = 0), 2SAF/weight (W = 2).

Fig. 9: Using 2RRAM devices per weight. The weights are adjusted based on the mapping method [26], to the set of values {-2, +2}

.

Therefore, this fact is exploited to develop a design technique, that chooses an activation function by considering its fault tolerance potential. Figure 8(a), presents the fault-tolerant activation function selection flow. Three pre-trained instances of BNNs using the $Sigmoid$, $Tanh$ and $ReLU$ functions are considered, and their fault tolerance capability is evaluated by mapping them to a faulty crossbar array. It is noted that both the $Sigmoid$ and $Tanh$ function produce non-sparse models as their neurons almost always are activated. That is because the produced output ranges are (0, 1) and (-1, 1), respectively, with the output not having a zero value or having a zero with a very low probability. Hence, from a fault-tolerance point of view, all faulty neurons have the probability of firing, which can propagate faults to the consecutive layers. On the other hand, the $ReLU$ function, has the advantage of adding sparsity to the network, meaning that a big percentage of the neurons are not active, enabling to mask some of the faults. As a result, it is expected that the BNN using the $ReLU$ function to be more robust to faults than the BNNs which use the $Sigmoid$ and $Tanh$ functions.

C. *Redundancy and Weight Range Adjustment*

Redundancy is on of the common and widely used techniques for tolerating faulty devices. However, it has its own disadvantages, with the energy and area overhead being the primary drawbacks. Moreover, in some cases the redundant devices can be faulty as well which can potentially nullify the benefits of redundancy. In this mitigation technique, we propose to use hardware redundancy for tolerating faults not by remapping the weights on secondary devices, but for increasing the range of the stored weight, minimizing the impact of a defective device, as shown in Figure 8(b). In the evaluation setup (Section III) it is stated that each RRAM device represents the value of one weight. If a SAF is present, the mapped weight's value might flip from -1 to +1 and vise versa.

To realize the redundancy and weight adjustment based fault tolerance method, the architecture of the crossbar is altered; from having 1T1R to 1T2R devices in parallel. As a result, each weight is mapped to two RRAM (2RRAM) devices. In order for the weight mapping on the crossbar to be valid, the weight's range is also changed to {-2, +2}, as can be seen in Figure 9. In the scenario that one SAF exists per weight, the weight's value of {-2, +2} will go to 0 and not {+2, -2} (Figure 9b). Therefore, we now have 3 possible values that the weight can hold {-2, 0, 2}. Compared to having the weights mapped to {-1, +1}, the extra state of 0 makes the output sum of the neurons, closer to the fault-free result. Of course, if high number of devices are faulty, the probability of having 2 SAFs per weight, is higher. In this case, the value of the weight would flip from $\pm$ 2 to $\mp$2, which is the same as the 1R device case. This observation is confirmed by our experimental results presented in Figure 11.

D. *Retraining for fault-tolerance in BNNs*

Retraining has been a widely adopted technique to compensate for the accuracy degradation of neural networks, due to various factors, such as hardware faults, transient errors, change in input features or operation conditions [29], [30]. For neural networks with online/on-chip learning features, retraining can restore the accuracy reduction with minimal effort. On the other hand, for networks with offline training, retraining techniques needs additional effort as the retrained weights have to be re-mapped to the hardware. In this case, retraining can improve the network's accuracy significantly, when combined with the fault-tolerance techniques presented in this paper, as shown in Figure 8(c). Therefore, the proposed framework exploits the potential of retraining to enhance the inference accuracy of BNNs mapped to faulty RRAM crossbars. In order to have efficient retraining results, the fault distribution of the crossbar array must be extracted and used by the retraining algorithm to exclusively train the non-faulty weights. Thus, the first step of retraining is to initialize the corresponding weight matrix according to the extracted fault distribution. Then, a gradient-mask is applied to exclusively focus on retraining the weights mapped to non-faulty hardware as shown below.

$$gradient\_mask = \begin{cases} 0, & \text{if weight} = SAF \\ 1, & \text{if weight} = fault - free \end{cases}$$

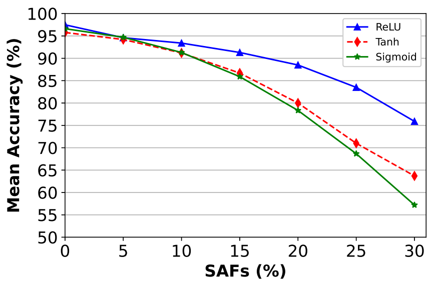
By using this mask we make sure that the retraining phase updates the fault-free weights only. The usage of the gradient mask enables the network to converge faster as it has less features to update, which can be translated to faster retraining time.

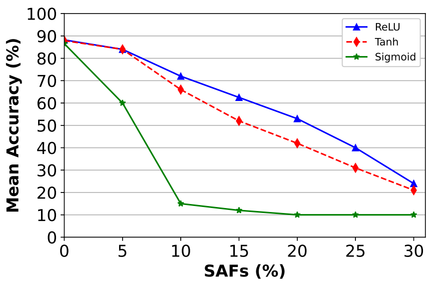
V. Results

The effectiveness of the proposed techniques is evaluated using the architectural and simulation setups presented in Section III-A. This section presents the accuracy improvement results of the proposed techniques. First the results of the fault tolerant activation function are discussed, then the redundancy and retraining techniques are applied on top of it, to further restore the inference accuracy.

A. *Fault tolerant activation function*

The fault tolerance capability of the three most commonly used activation functions, namely $ReLU$, $Tanh$ and $Sigmoid$ is evaluated and compared using the MNIST and Fashion-MNIST datasets as shown in Figure 10. As it can be seen from the plots in Figure 10, our insight on varying fault tolerance


(a) MNIST dataset


(b) Fashion-MNIST dataset

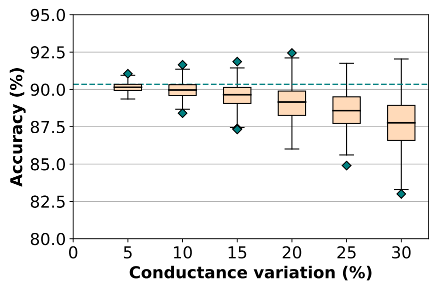
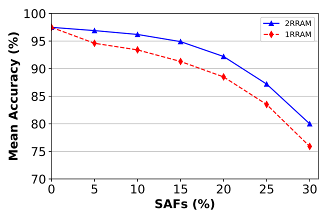

(c) Variation + 15% SAF on 2-layer ReLU MNIST

Fig. 10: Evaluation of SAF and conductance variation impact on the inference accuracy using different activation functions.


(a) MNIST dataset

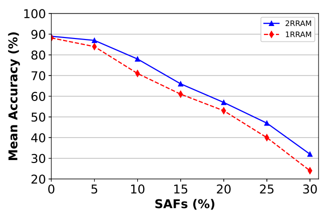

(b) Fashion-MNIST dataset

Fig. 11: Impact of SAFs using redundancy and weight adjustment technique

TABLE II: Retraining with SAFs

MNIST dataset			**Fashion-MNIST dataset**		**CIFAR dataset**	
SAF distribution (%)	Accuracy (%)		Accuracy (%)		Accuracy (%)	
	Baseline	Retrained	Baseline	Retrained	Baseline	Retrained
0	97.3	97.3	88.22	88.22	90.09	90.09
5	96	97.3	84	87.1	65	89.9
10	95.4	97	71	88	10	89.7
15	93	97.2	62	87.9	10	89.6
20	89	97.1	53	88	9	89
25	84	97.2	40	87.76	8	88.7
30	71	96.8	22	88.5	8	89

capabilities of different activation functions is validated from the simulation results. Figure 10(a) shows that both the $Tanh$ and $Sigmoid$ function exhibit similar accuracy reduction for the MNIST dataset across different rates of injected faults, while the $ReLU$ function leads to a minimal accuracy reduction. Hence, using $ReLU$ instead of $Tanh$ or $Sigmoid$ functions can yield up to 10% improvement without any additional fault tolerance technique. Similarly, in the 4-Layer BNN running on the Fashion-MNIST dataset shown in Figure 10(b), the $ReLU$ and $Tanh$ activation functions led to comparable accuracy reduction, while the accuracy for the $Sigmoid$ function drops drastically. It should be noted that, for 10-25% SAF distributions the $ReLU$ function has 5-10% accuracy improvement over $Tanh$. In general, the $ReLU$ activation function gives better results in terms of accuracy, across a wide fault range and for varying network architectures and evaluation datasets. Figure 10(c) also demonstrated the accuracy enhancement of the $ReLU$ function, in the presence of SAF and conductance variation.

B. Redundancy and Weight adjustment

The accuracy improvement potential of the proposed redundancy and weight range adjustment is also evaluated using the fault-tolerant activation function, $ReLU$ in this case, for the MNIST and Fashion-MNIST datasets. The improved accuracy results of the redundancy technique for the MNIST and Fashion-MNIST, are presented in Figure 11. For the 2-Layer BNN using the MNIST dataset (Figure 11(a), redundancy led to a high accuracy (>95%), when up to 15% SAFs are inserted. For higher SAF rates (20-30%), the accuracy is reduced gradually, while maintaining a 5% accuracy improvement. Regarding the 4-Layer BNN on the Fashion-MNIST, Figure 11(b), the accuracy enhancement of the proposed redundancy and weight adjustment method, varies depending on the rate of injected SAFs. Hence, for SAF = 10% the redundancy approach improves the accuracy by ≈8% when 15% SAFs are injected. This accuracy gain drops slightly to about 5% with more than 15% SAFs.

C. Retraining with SAF distribution

Table II presents the accuracy improvement results of the retraining method using the $ReLU$ activation function for different datasets. From the table it can be observed that the retraining method is able to almost fully recover the accuracy back to the baseline fault-free accuracy (SAF 0% entry in the table), for all datasets and SAF distributions. In the case of the MNIST dataset for a 2-Layer BNN, we can observe that even for a fault distribution as high as 30% the accuracy can be improved from 71% to 96.8%, which is almost equal to the baseline accuracy (97.3%) of the network with 0% SAF. For the Fashion-MNIST, the retraining method is able to recover the accuracy from a very low value of 22% up to 88.5%, which is almost ideal. Similar improvement is achieved for the CIFAR-10 dataset on VGG BNN, in spite of the complexity of the VGG BNN network.

D. Comparison with state-of-the-art techniques

In Table III, a comparison to other state-of-the art fault tolerance techniques [5], [4], [31] is presented. The comparison is conducted using a 2-Layer NN for the MNIST dataset, while having 20% SAFs. In the table, the term recovered accuracy is the ratio obtained when the restored accuracy is divided by the baseline accuracy, and the redundancy column (R) refers to the extra RRAM devices for representing the weights. The proposed technique in [5], uses a modified retraining method to incorporate the RRAM defects during the training phase, similar to the technique proposed in this work. While [5] recovers the classification accuracy up to 95.1%, our work restores the inference accuracy up to 99.8%. Authors in [4] propose a mapping algorithm (MAO), in order to map the weights on the RRAM based crossbar, that minimizes the mapping error. MAO recovers the accuracy up to 43%, without any hardware overhead. However, to achieve better results a redundancy scheme is utilized (R=2), resulting in recovering the accuracy up to 96%. In [31], the authors use different matrix transformations are used to tackle the impact of SAFs, combined with extra RRAM devices (R=2) for the weight mapping. When no redundancy is used, the recovered accuracy remains low (30.1%), whereas for R=2 the accuracy is restored up to 97.6%. Both methods [31], [4] add software overhead for implementing the mapping algorithm and the matrix transformations, while imposing also hardware overhead (R=2), for a better accuracy. Overall, our proposed fault tolerance framework outperforms the related works, restoring the accuracy up to 99.8%, with a comparatively less overhead.

TABLE III: Comparison with related fault tolerance techniques, using a 2-Layer NN for the MNIST dataset, with 20% SAFs.

Similar Works	**Recovered Accuracy (%)**	**Retraining**	**Redundancy (R)**
[5]	95.1	Yes	1
[4]	43	No	1
	96		2
[31]	30.1	No	1
	97.6		2
Our proposal	99.8	Yes	1

VI. Conclusion

RRAM crossbar array based Computation-In memory is a promising solution to overcome the limitations of traditional computing systems and deliver high energy-efficient computation path for emerging data and computation intensive application segments, such as neuromorphic computing. However, the widespread applicability of CIM is limited due to various reliability and variability challenges, and manufacturing difficulties. In this work, we investigated the impact of RRAM defects on the inference accuracy of BNNs and proposed different mitigation techniques to alleviate their impact. Results using multi-layer BNNs for the MNIST, Fashion-MNIST and CIFAR-10 datasets demonstrated that the proposed techniques can improve the inference accuracy in the presence of RRAM defects by up to 20%, 40% and 80%, respectively

References

[1] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor, “A classification of memory-centric computing,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 2, pp. 1–26, 2020.

[2] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, “Metal–oxide rram,” *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[3] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, “Accelerator-friendly neural-network training: Learning variations and defects in rram crossbar,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 19–24.

[4] W. Huangfu, L. Xia, M. Cheng, X. Yin, T. Tang, B. Li, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang, “Computation-oriented fault-tolerance schemes for rram computing systems,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 794–799.

[5] C. Liu, M. Hu, J. P. Strachan, and H. Li, “Rescuing memristor-based neuromorphic design with high defects,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[6] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang, “Fault-tolerant training with on-line fault detection for rram-based neural computing systems,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[7] W.-Q. Pan, J. Chen, R. Kuang, Y. Li, Y.-H. He, G.-R. Feng, N. Duan, T.-C. Chang, and X.-S. Miao, “Strategies to improve the accuracy of memristor-based convolutional neural networks,” *IEEE Transactions on Electron Devices*, vol. 67, no. 3, pp. 895–901, 2020.

[8] L. Xia, W. Huangfu, T. Tang, X. Yin, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang, “Stuck-at fault tolerance in rram computing systems,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 102–115, 2017.

[9] A. Chaudhuri, B. Yan, Y. Chen, and K. Chakrabarty, “Hardware fault tolerance for binary rram crossbars,” in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[10] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, “Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme,” *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2015.

[11] S. Hamdioui *et al.*, “Memristor for computing: Myth or reality?” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 722–731.

[12] A. Chen, “A review of emerging non-volatile memory (nvm) technologies and applications,” *Solid-State Electronics*, vol. 125, pp. 25–38, 2016.

[13] M. A. Lebdeh, U. Reinsalu, H. A. Du Nguyen, S. Wong, and S. Hamdioui, “Memristive device based circuits for computation-in-memory architectures,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[14] J. Borghetti *et al.*, “‘memristive’ switches enable ‘stateful’ logic operations via material implication,” *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[15] L. Xie *et al.*, “Scouting logic: A novel memristor-based logic design for resistive computing,” in *ISVLSI*, 2017.

[16] J. Yu *et al.*, “Enhanced scouting logic: a robust memristive logic design scheme,” in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 2019, pp. 1–6.

[17] I. Giannopoulos *et al.*, “In-memory database query,” *Advanced Intelligent Systems*, 2020.

[18] S. Hamdioui *et al.*, “Memristor based computation-in-memory architecture for data-intensive applications,” in *DATE*, 2015.

[19] R. Bishnoi, L. Wu, M. Fieback, C. Münch, S. M. Nair, M. Tahoori, Y. Wang, H. Li, and S. Hamdioui, “Special session–emerging memristor based memory and cim architecture: Test, repair and yield analysis,” in *2020 IEEE 38th VLSI Test Symposium (VTS)*. IEEE, 2020, pp. 1–10.

[20] I. Chakraborty, M. F. Ali, D. E. Kim, A. Ankit, and K. Roy, “Geniex: A generalized approach to emulating non-ideality in memristive xbars using neural networks,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[21] G. Charan *et al.*, “Accurate inference with inaccurate RRAM devices: A joint algorithm-design solution,” *Journal on Exploratory Solid-State Computational Devices and Circuits*, 2020.

[22] J.-H. Lee *et al.*, "Exploring cycle-to-cycle and device-to-device variation tolerance in MLC storage-based neural network training," *Transactions on Electron Devices*, 2019.

[23] E. I. Vatajelu, P. Prinetto, M. Taouil, and S. Hamdioui, "Challenges and solutions in emerging memory testing," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 3, pp. 493–506, 2017.

[24] W. Zhang *et al.*, "Design guidelines of rram based neural-processing-unit: A joint device-circuit-algorithm analysis," in *DAC*, 2019.

[25] W. Shim, Y. Luo, J.-s. Seo, and S. Yu, "Impact of read disturb on multilevel rram based inference engine: Experiments and model prediction," in *2020 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2020, pp. 1–5.

[26] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[27] D. Misra, "Mish: A self regularized non-monotonic neural activation function," *arXiv preprint arXiv:1908.08681*, vol. 4, 2019.

[28] S. Sharma, "Activation functions in neural networks," *towards data science*, vol. 6, 2017.

[29] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[30] J. Deng, Y. Rang, Z. Du, Y. Wang, H. Li, O. Temam, P. Ienne, D. Novo, X. Li, Y. Chen *et al.*, "Retraining-based timing error mitigation for hardware neural networks," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 593–596.

[31] B. Zhang, N. Uysal, D. Fan, and R. Ewetz, "Handling stuck-at-faults in memristor crossbar arrays using matrix transformations," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 438–443.