Stuck-at Fault Tolerance with Emerging Technology RAM in the NeuroSim MLP Neural Network System

by

An Qi Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

After decades of technology advancements, benefits from conventional dimensional scaling and effective scaling such as strain and high-k gate dielectrics are diminishing. In the post-Moore era, interests gathers around emerging technologies with greater performance that silicon. One such emerging technology is Carbon Nanotube Field-Effect Transistors (CNFETs). CNFETs have the potential to offer a lower power, higher performance semiconductor technology compared to it's silicon counterpart. However, CNFET technology being an emerging technology that has not yet reached maturity are still subject to high fault levels and levels of process variation. These high fault levels mean CNFET processes are unsuitable for semiconductor fabrication as general purpose designs require very low fault rates. Attempts to use CNFET processes have required increasing CNFET transistor sizes, defeating the purpose of finding a technology to replace silicon and continue Dennard scaling. Some success using emerging technologies has been achieved by using these emerging technologies with fault tolerant applications, such as neural networks in machine learning.

In order to understand the impact of neural networks to process faults, this work analyses the effect of stuck-at faults in neural networks. The NeuroSim system is used which implements a 2 layer Multi-Layer Perceptron (MLP) Neural Network. These two layers contain weight values which are stored in two Static Random Access Memory (SRAM) units. Stuck-at faults are applied to the two SRAM units in various patterns. These networks are re-trained to account for the faults, where the resulting accuracy indicates the resilience of the neural network system to stuck-at faults. With the effects of the stuck-at faults understood, fault recovery techniques to mitigate the effect of the stuck-at faults are proposed and evaluated. In the worst case without any recovery technique, the network's accuracy drops from 93.77% to 23.37% at a high fault rate of 40%. The fault rate indicates the percentage of SRAM bits affected by stuck-at faults. Stuck-at faults cause a SRAM bit cell to only read out one value, either 0 or 1. With a recovery technique, the accuracy is improved to 88.08% at a fault rate of 40%.

Acknowledgements

I would like to thank my supervisor Prof. Lan Wei and mentor Dr. Amr Tosson for their guidance, advice, and feedback.

I would also like to thank colleagues I've worked with, and Prof. Arie Gurfinkel for having introduced Emacs to me which I have found to be useful for various tasks.

Finally I would like to thank my friends and family for their support throughout this journey, which I appreciate immensely.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

BNN Binary Neural Network 2

CNFET Carbon Nanotube FET viii, 2–4, 6–9, 18, 19, 70, 71

CNN Convolutional Neural Network 7, 9, 71

CNT Carbon Nanotube viii, 2, 6–8, 19–21

CVD Chemical Vapor Deposition 6

DIBL Drain Induced Barrier Lowering 1

DNN Deep Neural Network 7, 9, 71

EDP Energy Delay Product 2, 6, 7

FET Field Effect Transistor 1, 2

HO Hidden to Output 16

IC Integrated Circuit 2

IH Input to Hidden 16

LDM Low Dimensional Material 2

LSB Least Significant Bit 46

MLP Multi-Layer Perceptron viii, 9, 11–15, 18, 71

MNIST Modified National Institute of Standards and Technology 8, 9, 14

MSB Most Significant Bit x, 45–48, 52, 53, 55, 57–59, 61, 66–68

NVDLA NVIDIA Deep Learning Accelerator 71

RAM Random Access Memory 3, 8, 9

RRAM Resistive Random Access Memory 2, 9

SNM Signal to Noise Margin 2

SRAM Static Random Access Memory viii, ix, 3, 7, 9, 11, 15–23, 29, 33, 44–46, 49, 51, 52, 55, 57, 58, 61, 63, 64, 66–68, 70, 71

SS Subthreshold Swing 2

TPU Tensor Processing Unit 71

Chapter 1

Introduction

1.1 Motivation

Silicon technology scaling has been one of the primary drivers of semiconductor industry in the past several decades. This was described by Moore's law which predicted the doubling of the number of silicon transistors in chips approximately every 18 months [6]. However as silicon transistors are scaled to be smaller and thinner, numerous limiting factors have appeared as they are scaled down closer to the sub-100 nm regime [37, 17]. These issues include Drain Induced Barrier Lowering (DIBL), gate tunneling, lowered mobility of thin silicon devices which cause increased current leakage, a lower on/off current ratio, higher power consumption, and lower performance [17].

In order to overcome these scaling limitations in silicon transistors and extract more performance from silicon, alternative silicon transistor structures such as the Fin Field Effect Transistor (FET) have been introduced to allow Moore's law to continue [42]. FinFET devices wrap the gate around a raised fin channel in order to provide better gate control cover the channel region, reducing any short channel effects and thus leakage current. Non-planar FinFET devices have allowed silicon technology to reach the 5 nm node, but is likely unable to progress further due to large device parameter variations caused by short channel effects, weakened gate control, and so on [30]. To achieve the 3 nm node, the stacked nanosheet FET structure which replaces the fin of the FinFET with stacked sheets of silicon covered in dielectric material has been proposed [42]. Despite this, thinned silicon device channels are still limited by the reduced mobility of thin silicon channels beyond 3 nm [38]. Thus, alternative channel materials are required if smaller technology nodes are to be realized.

As a result, interest is gathering around alternative materials and emerging technologies which have better mobility than silicon at similar device sizes. An example of such a material that have been studied as an emerging technology is CNT used in CNFET [23, 29]. As a Low Dimensional Material (LDM), CNTs exhibits higher carrier mobility values while having a thin channel material, making it an attractive choice for devices scaled down beyond 3 nm.

CNFETs are one emerging technology that have been studied for some time and is a prime example of the progress of emerging technologies that has come the closest to large scale production [23]. Recent device demonstrations [23] have shown high performance CNFETs can be fabricated with a tunable number of tubes between 100 to 200 tubes/µm, the ultimate density required for high performance Integrated Circuit (IC) applications [35]. Five stage ring oscillators built from this process were able to achieve an oscillating frequency of up to 8.06 GHz, corresponding to a stage switching frequency of up to 80.6 GHz. Other work has demonstrated 5 nm and 10 nm CNFETs with a Subthreshold Swing (SS) of 73 mV/decade and 60 mV/decade respectively, where 60 mV/decade is the theoretical room temperature limit. When compared to a silicon FET of the 10 nm technology node, the CNFET at 10 nm has an Energy Delay Product (EDP) 36 times better than the silicon device while also having a gate delay approximately 4 times smaller.

Beyond device demonstrations, CNFET chips have been fabricated to demonstrate the progress of CNFET processes [12, 41, 11]. One example is a simple CPU [12]. It's transistor count totaled to around 14,000 transistors, still far below the number of transistors in a modern CPU due to the immature CNFET processes imposing design constraints. The CPU was designed with specific combinations of logic gates that are less susceptible to process variations to ensure the gates had a usable Signal to Noise Margin (SNM), as well as very large transistor widths to reduce the effect of variation and faults in CNTs. Another example that was fabricated was a neuromorphic or machine learning chip which makes use of the variations in CNFET drive current and Resistive Random Access Memory (RRAM) resistance to implement a hyperdimensional computing application [41]. This neuromorphic chip is comprised of 1,952 CNFETs at a length of 1 µm and 224 RRAMs, 16 CNFET logic stages, consumes an average power of 5.4 mW, and was able to achieve an accuracy of 98% when classifying input sentences between two languages after 256 training iterations. One other example using CNFETs implements a Binary Neural Network (BNN) which makes use of incrementing or decrementing accumulated values with binary weights of ± 1 or 0. [11]. It makes use of grey code arithmetic (which differs in only one bit when incrementing or decrementing values) accumulators to reduce the effect of missed timing faults between clock cycles on the accumulated value, as well as upsizing transistors. These techniques allow it to preserve 90% of it's projected EDP.

These chips however do reveal that CNFETs, despite being the most developed emerging technology, still suffers greatly from process variation and faults. These variations and faults prevent CNFETs from being used in most modern silicon device applications. However, there exists a class of applications which by nature are able to tolerate faults. These applications are neuromorphic or machine learning applications. In the neuromorphic chip example [41], although 78% of it's bits used for it's neuromorphic classifier application were affected by stuck-at faults it was still able to function correctly and achieve an accuracy of 98%. It is clear that neuromorphic applications are able to tolerate faults. With these fault tolerant applications it is still possible to make use of the benefits of CNFET processes while having the application mitigate the effect of the process faults.

1.2 Scope of this Thesis

While emerging technologies such as CNFETs may not be mature enough for standard computing applications which require precise operation due to their fault rates, they can be used for fault tolerant neuromorphic applications. In particular we focus on stuck-at faults in this thesis. This work is divided into the following sections.

1.2.1 Analyzing the Effect of Stuck-at Faults in RAM on Functional Performance of a Neuromorphic Application

In order to determine whether emerging technologies can be used with fault tolerant neuromorphic applications, the effect of faults is to be measured and understood. This section analyses the effect of a class of faults in Random Access Memory (RAM) on the functionality of a hardware system which implements a neuromorphic application. This is to particularly measure the resilience of RAM units used for the neuromorphic application.

A neuromorphic hardware system is selected for analysis with faults injected to it's RAM units. The selected hardware system makes use of SRAM cells which is where the faults are applied for analysis. The impact of the faults on the SRAM cells of the hardware system is measured under various fault patterns. Appropriate changes to the hardware system's simulator are made in order to represent the faults and generate various fault patterns for analysis. From the resulting performance of the neuromorphic application, the resilience of SRAM cells in the hardware system and the effects the faults have on the neuromorphic application is thus understood. This prepares a baseline for the next section.

1.2.2 Fault Recovery Techniques to Overcome the Effect of Stuck-at Faults in RAM Units of a Neuromorphic Application

With the effect of the faults on the performance and functionality of the hardware system implementing the neuromorphic application understood, recovery techniques are presented in order to overcome the effects of the faults on the application's performance. Mechanisms and changes to the simulated hardware are proposed to counter the effects of various fault patterns analyzed in the previous section. The mechanisms are simulated in the neuromorphic hardware system's simulator with the appropriate modifications to functionally model the behavior of the mechanisms. The resulting performance of the mechanism under relevant fault patterns in the hardware system is measured again to show it's ability or inability to overcome particular fault patterns. Performance measurements from the previous section are used as a baseline to measure the relative performance improvement on the application. A recommendation is then made based on the resulting performances of the mechanisms against various fault patterns as well as their estimated overhead requirements.

1.3 Thesis Outline

The organization of this thesis is as follows. Chapter 2 discusses the state of emerging technologies, particularly CNFETs and their process variations which lead to faults. The specific class of faults this work analyses is also discussed. With the emerging technologies established as faulty immature technologies, fault tolerant applications are discussed. Arguments for using faulty emerging technologies with fault tolerant neuromorphic applications are then presented. Chapter 3 provides a background overview of a hardware system which implements a neuromorphic application that will be used for analysis with faults. An overview of machine learning details the neuromorphic application uses is also provided. In Chapter 4, the particular class of faults under analysis are applied to the neuromorphic application's hardware. Simulations are run training the neuromorphic hardware with the faults and the accuracy of the application is used as a baseline to measure the impact of various fault patterns. The effects of the faults on the accuracy are also analyzed and described. The changes to the hardware simulation are briefly described. Chapter 5 presents fault recovery techniques in order to overcome the effects of the various fault patterns. The accuracy improvements of these techniques against relevant fault patterns are shown and discussed. Additional changes made to the simulator are also discussed for each fault recovery technique. A recommendation of which fault recovery technique to use is made

depending on the fault pattern encountered, and overhead costs. Chapter 6 concludes the thesis with a summary and an overview of future work.

Chapter 2

Literature Review of Faults in Emerging Technologies and Neuromorphic Applications

This chapter presents variation and fault sources in CNFETs to outline a class of faults present in emerging technologies. This is followed by an overview of work in neuromorphic applications. With both fault prone emerging technologies and fault tolerant neuromorphic applications discussed, literature around applying emerging technologies to fault tolerant neuromorphic applications is presented. After describing the robustness of neuromorphic applications to faults, an overview of the work of this thesis is described.

2.1 Variation and Faults in Emerging Technologies

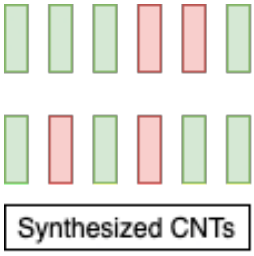
Emerging technologies such as CNFETs have been demonstrated to offer potential energy and performance benefits with their better carrier mobility and EDP compared to silicon. However, current immature fabrication processes are subject to high levels of variations and faults.

The fabrication process of CNTs for CNFETs starts with using Chemical Vapor Deposition (CVD) to grow CNTs. While these CNTs could be grown and used on a substrate, 33% of the resulting CNTs are metallic and do not function as semiconducting materials. Metallic CNTs act as short circuits, and degrade transistor performance parameters such as the on/off current ratio and the noise margin. [39] One technique used to remove the

metallic CNTs does so by applying high voltages to the CNTs in order to break down any metallic CNTs [28]. This technique however also breaks down a percentage of semiconducting CNTs as well, creating open circuit CNTs which reduce and thus vary the strength of transistors as well [39, 11]. The mainstream alternative to this process is to take the grown CNTs and disperse the metallic CNTs in a solution to separate metallic and semiconducting CNTs, attaining a collection of mostly semiconducting CNTs [18]. Various dispersion techniques are able to attain solutions with purities of more than 99.99% [35] and 99.9999% semiconducting CNTs [23]. These remaining metallic CNTs, besides degrading transistor performance, can also cause stuck-at faults in circuits. To illustrate, if a transistor contains a metallic CNT which acts as a wire and other transistors are not strong enough to control the node the metallic CNT affects, then the node is pulled towards the node at the other end of the metallic CNT. In the example of the cross-coupled pair in a SRAM cell, a metallic CNT placed between the output and a rail can cause the SRAM cell to be stuck-at 1 or stuck-at 0 [39]. Some work overcomes the effect of metallic CNTs by building CNFETs large enough to the point where the metallic CNTs only cause minor variations [12]. However, this is not ideal as this means that the emerging technology cannot be sized down to aggressive smaller technology nodes or benefit from using emerging technologies for their reduced EDP or improved performance. Figure 2.1 illustrates synthesized CNTs having most of their metallic CNTs sorted and removed before being placed in trenches to form a CNFET.

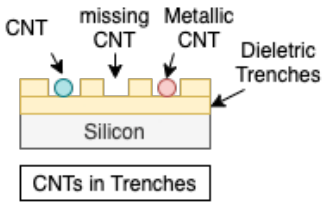
2.2 Neuromorphic Applications

Neuromorphic applications have progressed greatly in recent years thanks to the advances in silicon semiconductor devices improving the performance and power consumption of processors. These performance improvements allowed neuromorphic and machine learning applications to be run on CPUs and GPUs, spurring the growth of these applications. This has lead to the development and success of machine learning techniques such as Deep Neural Network (DNN), and Convolutional Neural Network (CNN). These neural networks solve challenging problems such as image classification, object detection, face detection, speech recognition, and so on [13, 24]. However, these applications are computation intensive and have high power requirements [15]. Running these applications on conventional processors is very time consuming and is power inefficient, so much so that custom chips and hardware systems have been designed to solely run these neural network applications [4, 15]. With these increasing performance and power demands, neuromorphic applications are a prime candidate that could benefit from the performance and power improvements emerging

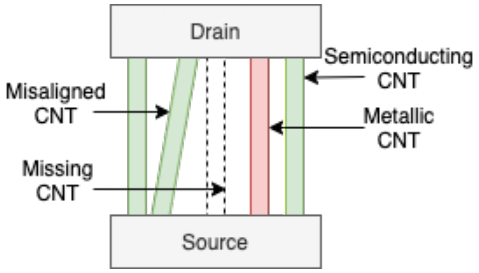


(a) Synthesized CNTs.

(b) Dispersed, separated, and sorted CNTs.



(c) Placed CNTs in trenches.



(d) Placed CNTs in a CNFET.

Figure 2.1: An example of CNTs sorted and placed in a CNFET. Green indicates a semiconducting CNT, red indicates a metallic CNT.

technologies can offer. In addition to the performance and power benefits, neural networks can potentially adapt to and recovery from faults, increasing their suitability towards faulty immature emerging technologies [32].

2.3 Tolerating Emerging Technology Faults with Error Resilient Neuromorphic Applications

As described in Section 2.2, neuromorphic applications are a potential candidate for use with emerging technologies for their performance and power improvements while tolerating the emerging technologies' faults. There have been chips designed using emerging technologies targeting neuromorphic applications [41, 22]. These designs have been able to achieve reasonable success in overcoming the effects of stuck-at faults from emerging technologies. One work fabricates a chip that classifies text between two languages with a high average accuracy of 96% while also improving the system level energy efficiency by a factor of 35 [41]. Another work makes use of non-volatile memory, an emerging technology in RAM cells, with a neuromorphic application that detects Modified National Institute of

Standards and Technology (MNIST) handwritten digits [22]. This design is able to recover 99.3% of it's base accuracy, with 20% of it's RAM cells subject to stuck-at 1 and stuck-at 0 faults.

2.4 The Work of this Thesis

Previous work in applying emerging technologies to fault tolerant neuromorphic applications have covered a variety of neuromorphic applications. One work mentioned in Section 2.3 uses hyperdimensional computing [41], a neuromorphic application. It achieves it's resilience to the high stuck-at fault rate of 78% in the hyperdimensional compute portion of it's hardware by reducing the complexity of the neuromorphic application to classifying 2 labels at a time, making it easier for the hardware to shift it's identified label to the correct label. Another work [22] uses dense MLP layers with RRAMs to recognize handwritten digits from the MNIST dataset, for a total of 10 classification labels to identify input digit images with [7]. It only tests random fault patterns applied to RRAM conductances where up to 20% of it's RRAM cells are stuck-at 1 or stuck-at 0.

This work analyses the effect of stuck-at faults in the dense layers of the NeuroSim hardware system which implements a MLP network to classify handwritten MNIST digits [3]. The layers are modeled as an array of SRAM cells which store the layer weights. Stuck-at faults in the CNFETs of the SRAM cells resulting from the variations of the emerging technology are considered in the SRAM cells. This is done in order to particularly measure and understand the resilience of memory units and by extension the layer weights in neuromorphic applications implemented in hardware. It is of note that dense fully connected layers are a common component of DNNs and CNNs. This work uses dense layers as opposed to [41] that uses hyperdimensional computation vectors and exclusive OR operations to label input sentences as one of two language labels, a simpler application which results in a simpler hardware design. These dense layers use digital weights of limited integer bit precisions which can be applied to other emerging technologies that implement weights with multiple integer bits. This is as opposed to [22] which uses RRAM conductances to model weights and is thus limited to conductance based memory. Furthermore the analysis is performed with various fault patterns beyond random fault patterns, where the effect of the faults on the network is described as well as opposed to [22] which only covers random faults. The fault rates explored go beyond the 20% that [22] covers and up to 40% instead, measuring the neuromorphic system's ability to maintain a reasonable accuracy at even higher fault rates and identifying the effect of higher fault rates on the network. Once the resilience against and effect of stuck-at faults is analyzed, four fault recovery techniques are

presented to improve the resilience of the memory in the neuromorphic system as opposed to [22] which presents one technique.

Chapter 3

NeuroSim Background

This section presents background material in the area of machine learning and the SRAM neural network system the NeuroSim simulator models.

3.1 Machine Learning

Machine learning consists of a variety of techniques to solve tasks which are difficult to solve through traditional algorithms. These variety of techniques are used to target various tasks. One task in particular is classification, where supervised machine learning techniques can be applied. Supervised machine learning techniques learn a function f that takes input x and produces output y. Classification is a problem of identifying which label to label a given input as, such as labeling an image of a numerical digit as 2 or 3. As such, supervised machine learning techniques can be used to learn a function f that takes a numerical digit image x to produce an output label of y.

This function f can be realized as a MLP network. The MLP network consists of layers of neurons, mirroring biological neurons where neurons are connected to other neurons. These connections are also referred to as synapses, and have a weight value representing how strong or important a connection is. For the MLP network, this connection takes input from a previous layer where the first layer takes input from the given input values. The given input values are weighed through the synapse weights. With the right combination of synapse weights at a neuron, certain features of the input can be identified by weighing them at higher importance levels. If the input has the features that the neuron weights are learned to recognize, then the neuron activates and signals to the next layer that it has

detected it's feature. Figure 3.1 illustrates this model. The x input values are multiplied by weights w The results are summed and provided to an activation function as input, where the activation function's output is taken as the output of the neuron.

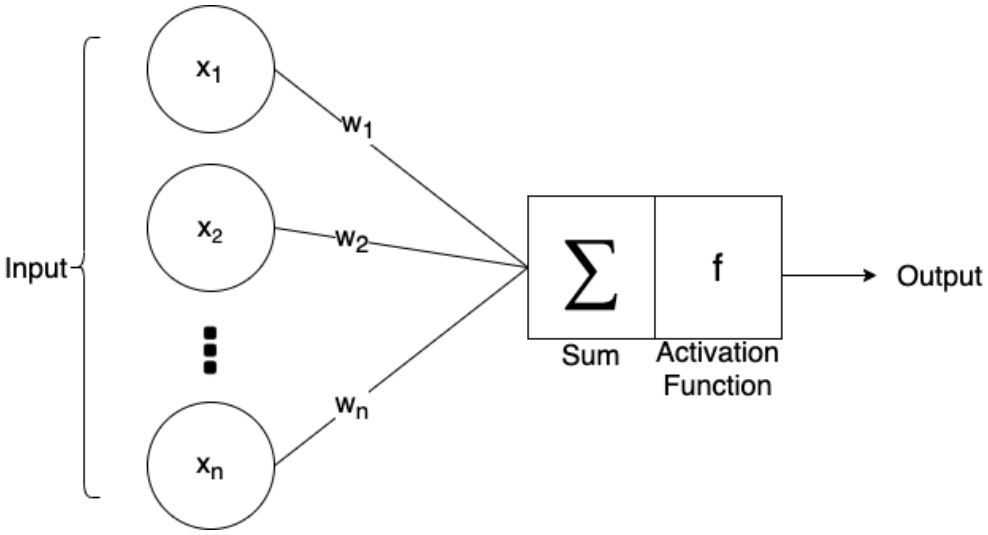


Figure 3.1: An example of a neuron.

In order to activate a neuron, a neuron sums the product of the input values multiplied by their corresponding synapse weights. The sum the products is used as the input to the neuron's activation function. The activation function maps the provided value to an activation value that reflects how greatly the neuron activates from the provided input. This activation function effectively takes a measure of the input multiplied by how important each part of the input is and converts it to a numerical value. This numerical value corresponds to whether or not the neuron has detected the feature is was trained to detect. This activation function can be a variety of functions, one in particular being the sigmoid function. Figure 3.2 shows an example of a sigmoid function.

The next layers perform a similar function, and have weighted synapse connections which are trained to recognize a combination of features from the previous MLP network layers and produce their own activation values. These layers in the middle of the network between the input and output are referred to hidden layers and their neurons are referred to as hidden units. At the last layer, or the output layer, the output of the neurons are used to determine the classified label by measuring which output neuron has the highest activation value. As it classifies an input to one of N values, there are N output units. The overall network can be seen in Figure 3.3. The neurons from Figure 3.1 form the hidden

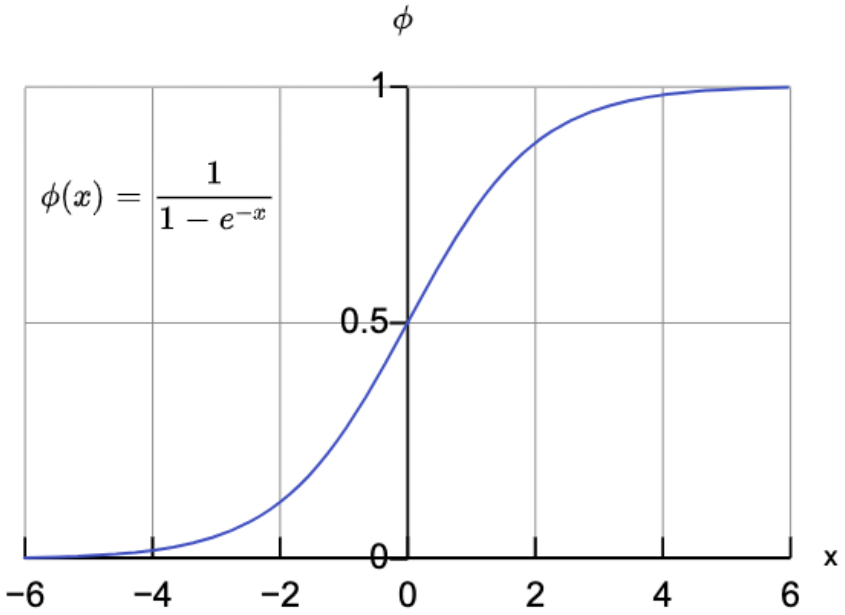


Figure 3.2: An illustration of the sigmoid function.

layer and output layer. The input layer represents the input values provided to the hidden layer. The input to hidden layer synapse weights are represented by the connecting arrows from the input to hidden layer. The output layer is similar, and receives it's input from the hidden layer. These input values are multiplied with it's hidden to output synapse weights.

The function f's MLP network weights are learned after being trained on example input data which is provided as the input x to produce output y. To learn the weights of f, an optimality score representing how accurate the function f is can be computed from the output y. This score is referred to as the loss from a computed loss function and can be seen in the equation below, where t is the target value, y is the output value, and L is the loss.

$$L = (t - y)^2 \tag{3.1}$$

This loss function is minimized over multiple training iterations by using a gradient descent algorithm. Particularly, the stochastic gradient descent algorithm is used. The stochastic gradient descent algorithm randomly samples subsets of the training data for the gradient descent algorithm. The gradient descent algorithm differentiates the loss and

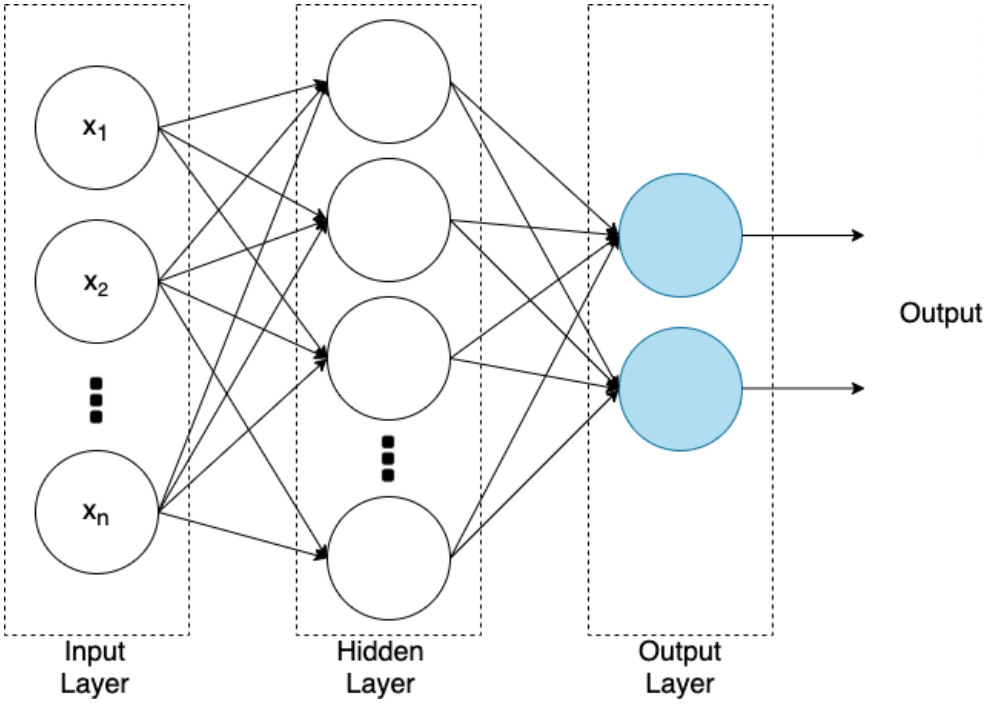


Figure 3.3: An example of a MLP network.

pushes the weights towards more optimal values by adding the negative of the differentiated slope to ideally reach the lowest loss value.

3.2 NeuroSim's MLP Neural Network

NeuroSim models a MLP network similar to the general network model described in Section 3.1. The MLP network is designed to target the MNIST dataset. The MNIST dataset is a dataset of handwritten digits, where neural networks are designed to recognize and label these handwritten digits.

This MLP network consists of 2 layers, as seen in Figure 3.4. An input to hidden layer, and hidden to output layer. The first layer takes the MNIST digit images as input and detects their features. The second layer detects if a combination of features is present, and generates a score of how likely the digit image is for each of the 0 to 9 possible digit labels.

The digit images from MNIST are 28 by 28 grey-scale digit images. The digit images have 4 white-space pixels along their edges. These white-space edges in the digit images

are cropped to 20 by 20 to reduce the amount of unneeded calculations. Thus the first layer consists of 400 or 20 by 20 input units. In the middle, the base network consists of 100 hidden units. Corresponding to the 10 possible digit labels it can label the input as of 0 to 9, there are 10 output units. Furthermore, these grey-scale values double precision floating point values are further truncated to 1 or 0 "black" or "white" images. If the pixel is at or above the threshold of 0.5, then the pixel is set to 1. Otherwise it is set to 0. This simplifies the realization of the MLP network in NeuroSim.

The system realizes the MLP network by feeding in the digit images pixel by pixel. Since each pixel of the 20 by 20 pixels corresponds to one of the 400 input units which are multiplied by a weight, the weights of each of the 100 hidden units can be read out for usage column by column for each of the input pixels.

Thus the layers are realized as memory arrays, with each column representing a hidden unit or output unit for the input to hidden layer and the hidden to output layer respectively.

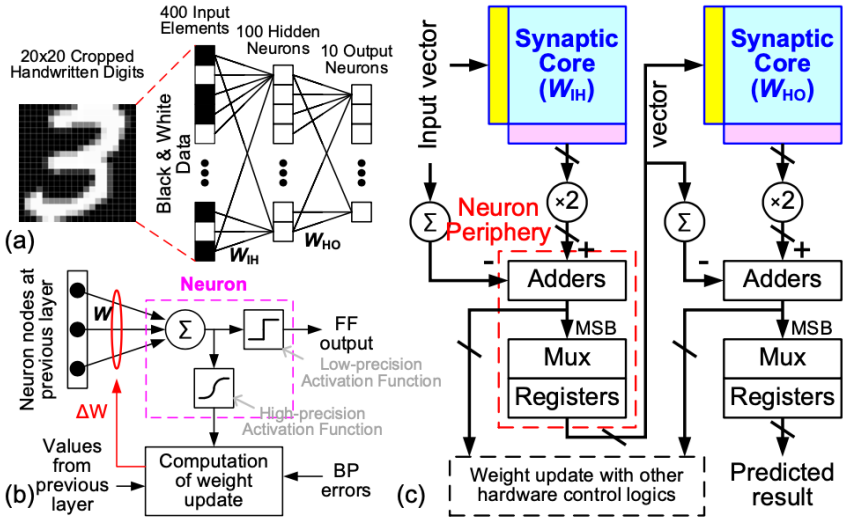


Figure 3.4: The overall NeuroSim system [5].

The NeuroSim model that will be used is the SRAM model, where each weight is represented by multiple SRAM cells. These weights are now represented by 6 bit integers in the base network.

Additionally NeuroSim maps the weights from it's digital 6 bit values to the -1.0 to 1.0 range for it's higher precision computations in it's training backpropagation step.

To map these weights onto a -1.0 to 1.0 value range, the weights are can be thought

of as representing values from a 0.0 to 1.0 range, where 0 is the minimum value a weight can take, and 1 is the maximum value can take. Note that for the base 6 bit case, the maximum value is 63. These values can be mapped the values to the -1.0 to 1.0 range by subtracting by 0.5, or half the maximum value, and multiplying by two. To simplify the implementation NeuroSim instead subtracts the maximum value from double the input and weight product, as seen in Figure 3.4. Additionally, since the input is limited to 0 or 1, the multiplication is not necessary.

NeuroSim also converts the sigmoid activation values to digital 0 or 1 values. The sigmoid is effectively digitized by outputting a 1 if the sum of the inputs reaches at least 0 in terms of input, or 0.5 in in terms of output in the sigmoid activation function. This eases the implementation by keeping the input to hidden layer and hidden to output layer consistent.

The columns of each SRAM array represent an output unit of the layer it corresponds to. This can be seen in Figure 3.4 as the blue synaptic cores, labeled as the Input to Hidden (IH) layer and Hidden to Output (HO) layer. The rows of each SRAM array correspond to an input unit from the previous layer, or the input in the case of the input to hidden unit layer. The sigmoid activation of the output units of the layers can easily check if they are beyond the threshold by checking if the final sum is greater than or equal to zero. The weights of each column per row are summed if the incoming pixel from the left is 1. Otherwise the summation operation can be skipped. The result of the summed values from the adders can be checked whether they're positive or negative to implement the digital, low-precision, activation function. The digital outputs are then used as input to the next layer.

Figure 3.5 shows the contents of a synaptic core SRAM array. multiple SRAM cells are used to represent a N bit weight value. These SRAM cells are the cells which will have stuck-at faults applied to them for analysis.

Equation 3.2 illustrates the output value calculation y_k for each of the $k = 10$ different output label values. The variables w_{ij} and v_{jk} are the weights of the input to hidden layer and hidden to output layer respectively. The x_i represents the input digit images' pixels which are set to 0 or 1. As shown in the inner parenthesis, the input digit pixels are multiplied by weights of the input to hidden layer. The multiplied values are summed and used as input to the sigmoid activation function for each of the $j = 100$ hidden units, j being the number of hidden units. Moving to the outter parenthesis, this vector of 100 hidden unit activation values is used as input to the hidden to output layer weights. The summed value for a given output unit k is then also used as input to the sigmoid activation function. The output unit value with the largest value is then taken as the digit label to

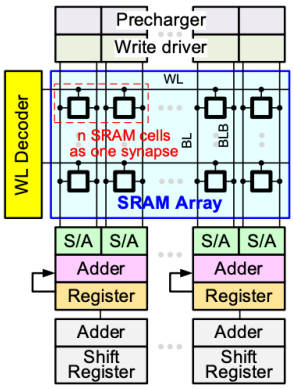


Figure 3.5: An SRAM array used in NeuroSim [5, 3]

label the input digit image with.

$$y_k = \sigma\left(\sum_j v_{jk}\sigma\left(\sum_i w_{ij}x_i\right)\right) \tag{3.2}$$

Figure 3.6 illustrates this calculation in NeuroSim. An input image of a digit is shown. In this example, the digit is 7. The black parts indicate the pixel value is 0, and white indicates 1. As the input digit image is 20 by 20 pixels, the length of the input vector i is 400. The pixels are multiplied by weights which identify particular features of the digit. In this case, the example hidden unit shown checks for the rising arm of a 7 as well as the end tip of the 7. The weights of the example hidden unit in the w_{ij} weight array is highlighted by the red column. The 1 and 0 pixel images are multiplied by weights which are mapped to the -1 to +1 range. If the sum is at least 0 or positive, then the digital sigmoid produces an output of 1 for use in the next layer. Otherwise, it outputs a 0. As this hidden unit particularly checks for a feature of the digit 7, this hidden unit outputs a 1 for this input image of 7. A cursory glance at the overlaid 7 and the hidden unit weights also indicates the majority of the 7 is covered by weights mapped to 1. The output vector of the input to hidden layer digital activation values is sent to the hidden to output layer. Once again, the input vector, or the hidden unit output vector in this case, is multiplied by a set of weights. As the input represents the detection of certain features in the input digit image, the hidden to output layer weights checks for a certain combination of digit features. For example, the output unit 7 would have high weight values for any features that exist in the digit 7, and low weights for other features. Here, the summed values across the product of the detected hidden unit features and their weights for the digit 7 should be the highest

for the network to label 7 as the input digit.

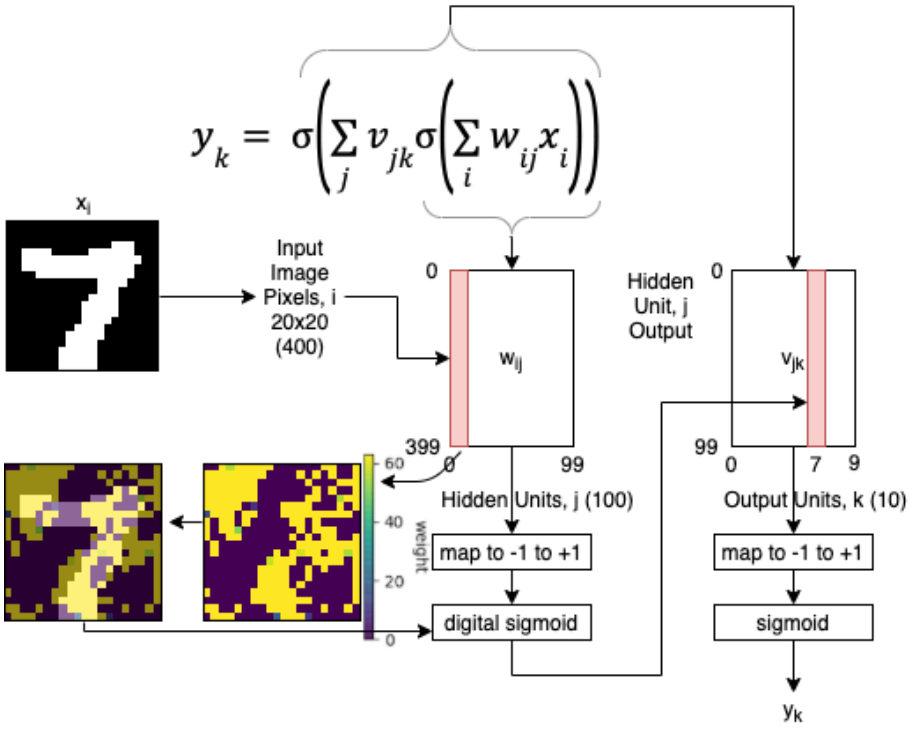


Figure 3.6: An example the MLP system calculating an output likelihood value of an input digit.

3.3 CNFET Emerging Technology for NeuroSim and Stuck-at Faults in SRAMs

The main purpose behind using CNFETs with NeuroSim is that an emerging technology should provide a neuromorphic application with a lower power and higher performance technology to realize it with. The faults of the emerging technology would be absorbed or handled by the neuromorphic application as described in Section 2.3. Stuck-at faults are applied to NeuroSim's SRAM arrays to determine to what level of stuck-at faults can the system's memory handle and still have a reasonable functioning accuracy as described in Section 2.4. This is to test the ability of NeuroSim's weight SRAM arrays to tolerate high levels of stuck-at faults that arise from immature emerging technologies, effectively

performing a worst case analysis. This worst case analysis is attributed to not meeting the >99.9999% semiconducting CNT purity with a state-of-the-art process [23] in some fabricated devices, CNTs placed as misaligned or missing, and the metallic CNT removal process [28] being largely unsuccessful or removing more semiconducing CNTs than expected.

As described in Section 2.1, the process of fabricating CNFETs is subject to variation and faults due to CNT faults. These CNT faults arise due to metallic CNTs and missing CNTs from the metallic CNT removal process. Missing CNTs weaken the drive current of the transistor and metallic CNTs result in a transistor that is stuck on, similar to a wire. In an SRAM cell consisting of a cross coupled inverter pair, a transistor that is stuck on will result in the SRAM cell being stuck-at a value of either 1 or 0. Figure 3.7 illustrates an example of a stuck-at 1 fault due to a metallic CNT in an SRAM cell. The metallic

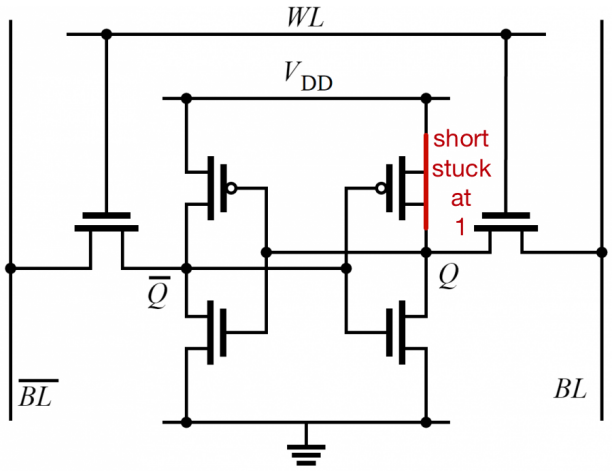


Figure 3.7: A stuck-at 1 fault in a SRAM cell due to a metallic CNT, marked in red.

CNT marked in red leaves the transistor stuck-on, shorting the cell's read out node to the rail [39]. When the cell is read from it reads a value of 1.

While there are no extensive studies of SRAM stuck-at fault patterns with numerous SRAM arrays fabricated at aggressive sub-100 nm nodes after metallic CNT removal and breakdown procedures, the distribution of stuck-at faults in SRAM cell arrays presumably take either of the following general patterns in order to account for a variety of fault patterns. One fault pattern the arrays are considered with is a random fault pattern where the metallic CNTs and resulting SRAM stuck-at faults are distributed randomly similar to [22], where after the CNTs are randomly dispersed for separation of metallic and

semiconducting CNTs [2] and most metallic CNTs or metallic CNT sections are removed, the remaining metallic CNTs cause stuck-at faults randomly across the SRAM arrays.

The other class of fault pattern the arrays are considered with are clustered fault patterns, where the stuck-at faults are close to each other. This arises when metallic CNTs are deposited in close proximity to each other and survive the removal process, or a segment of metallic CNT lands across multiple transistors affecting multiple SRAM cells and is not removed [19, 20], or a combination of the two. Overall this results in stuck-at faults that are in close proximity to each other.

Chapter 4

Stuck-At Fault Analysis

To understand the effect of stuck-at faults in SRAM and whether or not they can be used even at high fault rates, the accuracy of the network is measured with the network re-trained with the stuck-at faults. The accuracy of the network without re-training is too low to be used, and thus re-training is required. In this chapter the changes for modeling faults in NeuroSim, what fault patterns are generated and evaluated, re-training changes in NeuroSim, and the resulting impact of fault patterns are described.

4.1 SRAM Array Stuck-At Fault Patterns

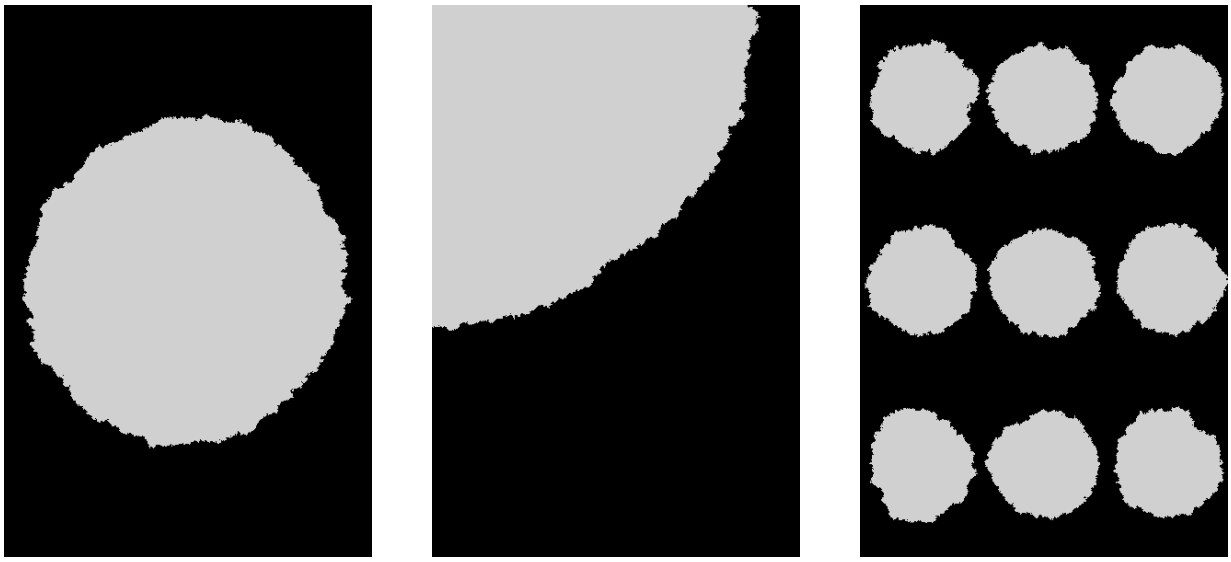
The fault patterns that are generated for analysis on the SRAM units are randomly distributed faults, and clustered faults. Across all these fault cases, fault rates of 10%, 20%, 30%, and 40% are used. These fault rates indicate what percentage of the total number of SRAM cells have a stuck-at fault. The randomly distributed fault pattern is a random assignment of stuck-at faults to the bits of the SRAM arrays, resulting from surviving metallic CNTs being distributed randomly as described in Section 3.3. The clustered fault pattern assigns stuck-at faults to SRAM cells which are adjacent to other SRAM cells that have been assigned a stuck-at fault already. This results in faults being assigned in a cluster shape, emulating the placement of surviving metallic CNTs in close proximity to each other.

Randomly distributed faults can be assigned randomly across the entire SRAM array being considered. However in the case of clustered faults, since the clustered faults form a cluster around a point, the positioning of the cluster is another factor that impacts

the stuck-at fault consequence analysis. In order to cover the majority of the different fault cluster varieties, the clustered fault patterns are analyzed with different fault cluster placements. An analysis is performed with the fault clusters positioned in each of the following scenarios, where Figure 4.1 illustrates an example of the clustered fault patterns.

1. A cluster placed in one of the 4 corners and the middle of each of the 2 SRAM arrays in NeuroSim. These 4 corners are the top left, top right, bottom left, and bottom right of the SRAM arrays. This results in 5 cluster analysis cases.

2. Consider each SRAM array to be divided into 3 by 3 grids, each with a cluster placed in the middle. Nine clusters are thus placed in each of the two SRAM arrays. This results in 1 cluster analysis case



(a) An example of a cluster of faults placed in the middle of an SRAM array.

(b) An example of a cluster of faults placed in the top left of an SRAM array.

(c) An example of a cluster of faults placed in the 3 by 3 segments of an SRAM array.

Figure 4.1: Examples of stuck-at faults applied in different clustered fault patterns. Each pixel represents a bit in the input to hidden layer. Black indicates no stuck-at fault at that bit, and grey indicates a stuck-at fault is present at that bit.

Thus the overall set of fault patterns that are analyzed are the random fault pattern, the single cluster fault pattern analyzed for each of the 4 corners and once in the middle,

and the 9 clusters centered in each SRAM array as though they were divided into 3 by 3 grids. This variety of fault patterns aims to cover the main variations of possible fault distributions for analysis. These fault patterns will be trained and analyzed with stuck-at 1 faults and stuck-at 0 faults separately so that the effects of each are understood individually. A fault pattern case that can be re-trained with an accuracy of at least 80% and up to a fault rate of at least 30% is considered to be fault tolerant.

4.1.1 Fault Pattern Representation in NeuroSim

The fault information is represented in a similar manner to how NeuroSim represents it's SRAM arrays, that is with a C++ object that stores information for the bits of the SRAM array. One object is created for each of the 2 SRAM arrays for the input to hidden layer and the hidden to output layer. As stuck-at faults are a bit level fault of the SRAM cells, they are represented by an enumerated value of either `NOT_STUCK_AT`, `STUCK_AT_1`, or `STUCK_AT_0`. A C++ vector of the size of the SRAM array's length by width stores one of these values for each bit.

To apply these faults to a provided weight, the stuck-at values are taken from the corresponding array layer's vector and combined with the weight. To combine the weight and fault, the fault values are converted into a form which allows for the fault to be applied to an integer weight. This is realized by performing a corresponding bitwise-or operation and a bitwise-and operation for stuck-at 1 faults and stuck-at 0 faults respectively. The stuck-at 1 values are read from the vector and are converted into an integer where the bits which are marked as `STUCK_AT_1` are set to 1, and all other bits are set to 0. Thus when a bitwise-or operation is applied to the weight, any bits which are not stuck-at 1 are left as they are and any stuck-at 1 bits are set to 1. Likewise, stuck-at 0 values which are read are converted into an integer which sets all bits marked as `STUCK_AT_0` to 0 and all other bits to 1. The bitwise-and now results in only setting the bits which are stuck-at 0 to 0, and all other bits are left as they are.

4.2 Training with Fault Patterns

In order to train the network with faults, the faults must be applied to the weight update delta in the training phase. This is particularly important due to the manner that the weights are mapped to value for the activation function, which in this case is the sigmoid function. The unsigned integer weights are mapped to -1.0 to 1.0 range with 0 as the

middle value of the maximum unsigned integer weight. Without applying the fault value to the weight, some weights which the network attempts to reduce the value for will have their weight value set to a negative value in the -1.0 to 1.0 range while the fault may increase the actual intended value to cross over to a positive value inside the range. If this happens then the update delta for the previous layer's weights may be updated with the wrong sign, pushing the weight to the opposite of the intended value.

4.3 Stuck-At 1 Fault Pattern Resilience Results

The base accuracy of the network without faults is approximately 93.77%. In general, the 10% fault rate is recoverable for most of the cases. From Figure 4.2, the middle, top left, top right cluster cases, and 3 by 3 cluster case performs the worst. The cases which perform the better are the random fault case, top right case, and bottom left case.

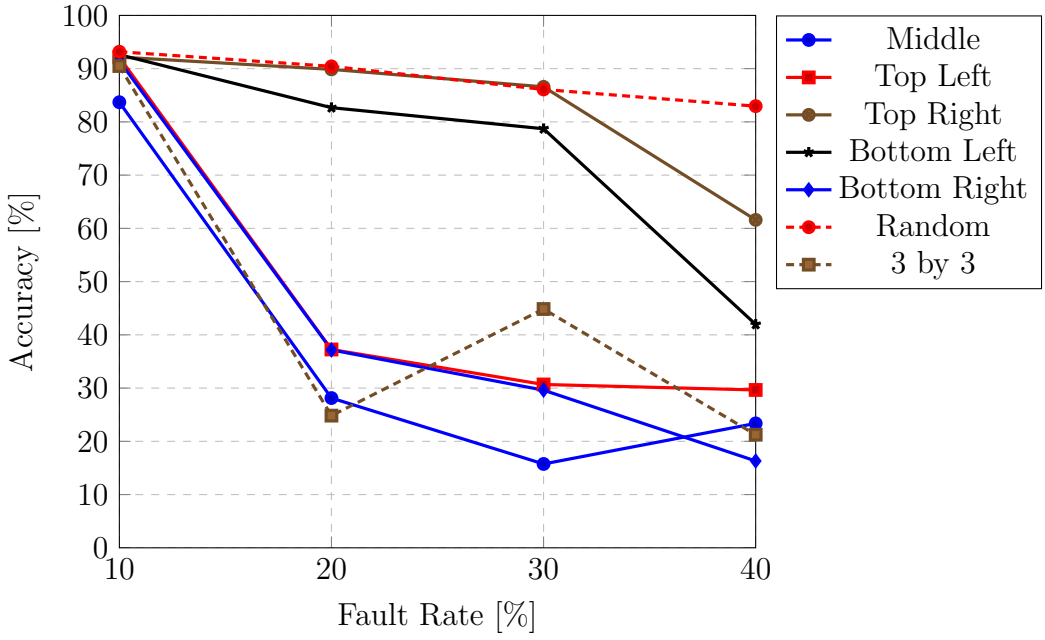
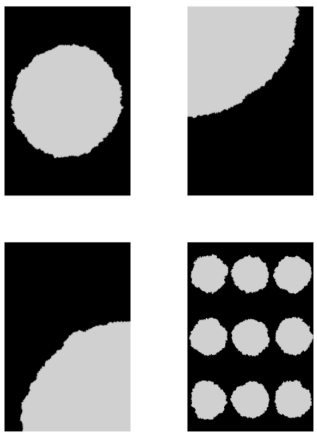


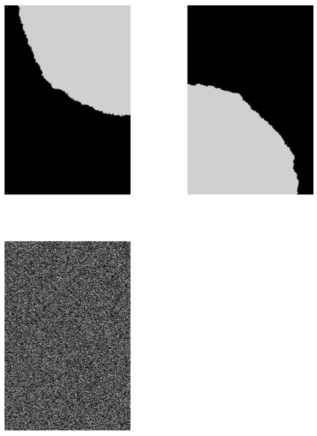
Figure 4.2: Accuracy after re-training the network under the middle, top left, top right, bottom left, bottom right, 3 by 3 cluster, and random fault patterns with stuck-at 1 faults.

For clarity the intolerable and tolerable fault pattern cases are shown in Figure 4.3. The intolerable cases with a sharp drop in accuracy at the 20% fault rate are shown in

Figure 4.3a. The tolerable cases that are able to maintain an accuracy of at least 80% up to the 30% fault rate are shown in Figure 4.3b.



(a) The intolerable stuck-at 1 fault pattern cases.



(b) The tolerable stuck-at 1 fault pattern cases.

Figure 4.3: The tolerable and intolerable stuck-at 1 fault pattern cases.

4.3.1 Random Fault Pattern Case

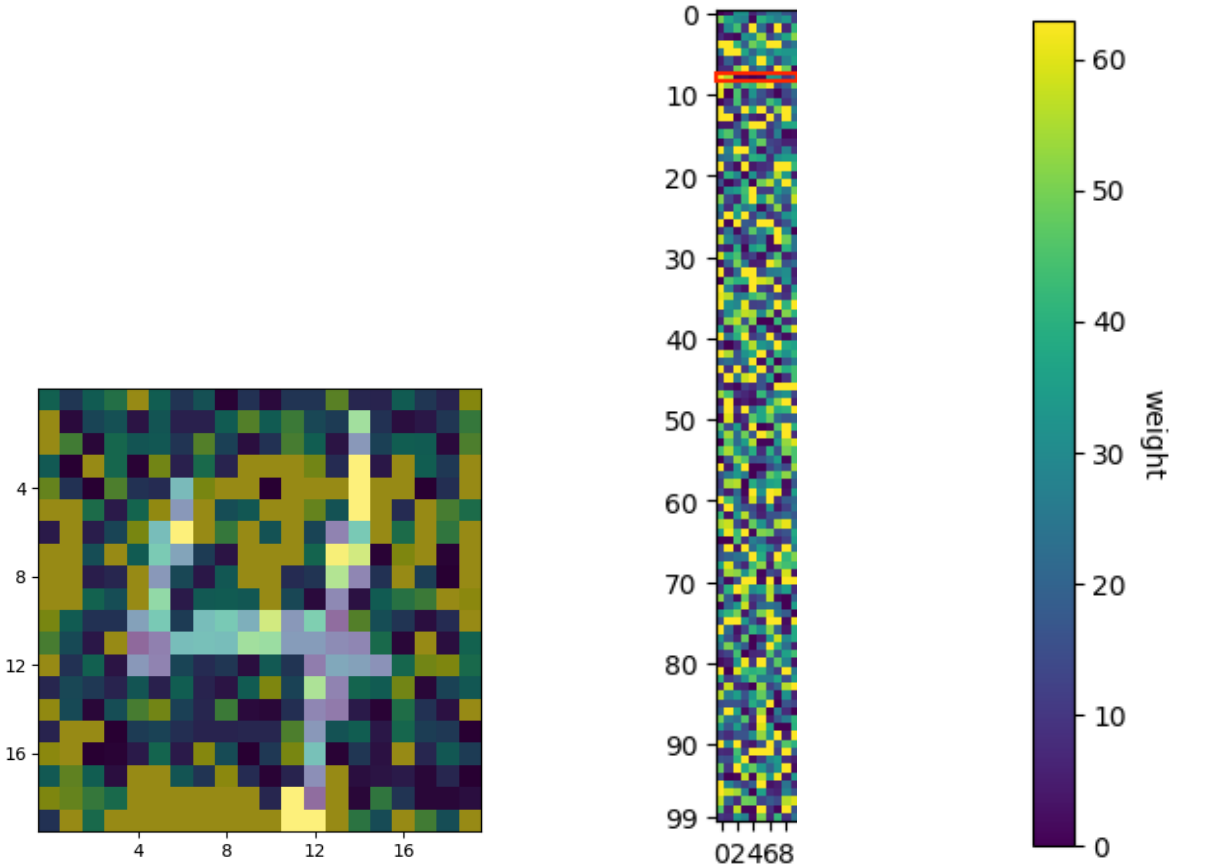
The random fault case performs the best in general. This is due to the weights still being usable, though in an unexpected manner. The network is still usable with 40% of the bits stuck-at 1 where the faults increase the value of many weights. Re-training with the random stuck at faults results in the hidden to output layer with a uniform distribution of weight values, where the sum of the weights of each column were roughly equal due to the uniform distribution of the random faults. The mechanism the network uses to make use of these weights is to perform the opposite operation which is normally performed. The input to hidden layer still detects features of digits, but now it rejects digits with features that it is searching for and does not activate upon encountering them. Additionally the hidden to output synapse weight is reduced as close to 0 as possible. Since the hidden to output layer weights of other columns corresponding to the hidden unit are likely to be raised to higher values due to the stuck-at 1 faults, the other columns will not have a high or medium value weight accumulated. As the sum of the weights of the columns are roughly equal, this effectively acts as subtracting a medium or high value weight from the other columns which do not match the features the hidden unit rejects.

An example is illustrated by Figure 4.4. Figure 4.4a shows an example of the digit 4 being rejected a hidden unit. The hidden unit, not accumulating a high enough value to activate the sigmoid function sends a value of 0 to the hidden to output layer. The columns for the other digits in the hidden to output layer in Figure 4.4b then cannot add the increased weight values due to the stuck-at 1 faults. In particular, the weights at the sides with higher values would not be added in the row marked in red. Figure 4.5 illustrates an example of a hidden unit rejecting the digit 2, and outputting a digital activation value of 0. This causes the high weights of other output units in the adjacent columns to not be added to their output unit's summed output value, reducing the chance of other units being selected as the digit label.

4.3.2 Fault Intolerant Cases

From Figure 4.2 the three lowest accuracy clustered fault cases were the middle, top left, and bottom right cluster cases. In these cases the fault cluster increases the chance a group of hidden units activate, where the input hidden unit connection weights of the hidden to output layer also have a fault cluster increasing their weights. The columns of the hidden to output layer correspond to selecting particular digits, the increased activations that activate cause the digit's column to accumulate more weight from the fault cluster. Thus the compounded error from the fault clusters result in digits covered by a fault cluster to be selected more often. An example of this can be seen in Figure 4.6. The input to hidden layer's hidden units 20 to 80 are fed to the hidden to output layer's weights. However, the hidden units 20 to 80 between these layers are both affected by stuck-at 1 faults in the layers. This results in many input digit images activating the middle hidden units 20 to 80, and subsequently causing the hidden to output columns to accumulate higher values, especially the center columns. This causes the middle digits to be mis-identified by the network more often.

Figure 4.7 shows an example of calculating the output values with the middle cluster in the MLP system. A input digit image of the digit 1 is used in this example. The majority of the hidden units j from 24 to 76 have a large portion of their weights covered by stuck-at 1 faults around the center of their 20 by 20 or 400 weights. As a result, the majority of the hidden unit output digital activation values corresponding to hidden units 24 to 76 are 1, as indicated by the white portion of the output vector in the figure. These 1 values are multiplied by the weights in the hidden to output layer. The weights corresponding to the output unit 5 for digit 5 are also affected by stuck-at 1, also covering a large portion of weights from 24 to 76 with stuck-at 1 faults. The stuck-at 1 faults greatly increase the weights as indicated by the yellow output unit weights between 24 to 76. As a result,



(a) A hidden unit's weights from the 40% random fault pattern case which rejects the digit 4 by purposefully not activating.

(b) The hidden to output layer which uses the rejected digit 4 to avoid accumulating weights from other columns. The row is marked in red.

(c) Weight value mapping bar.

Figure 4.4: An example of the network dealing with random stuck-at 1 faults at high fault rates by rejecting digit features to avoid accumulating weights in adjacent columns increased due to the stuck-at 1 faults.

the output value corresponding to the digit 5 is generally the highest and is frequently mis-identified by the network.

One point of note from Figure 4.2 is that the middle cluster case has a slightly increasing accuracy at 40%. This is due to the middle cluster being larger and expanding to cover more center weights so that the columns corresponding to digits 5 and 6 are more evenly

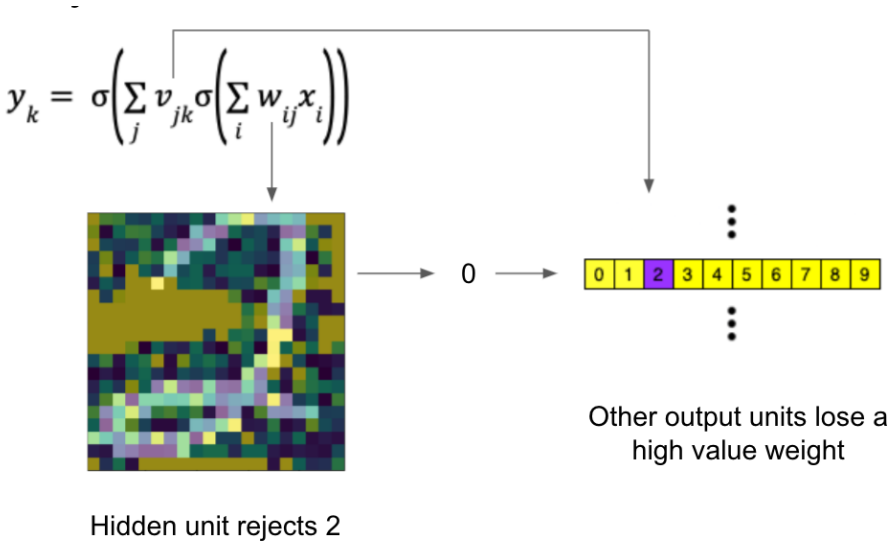
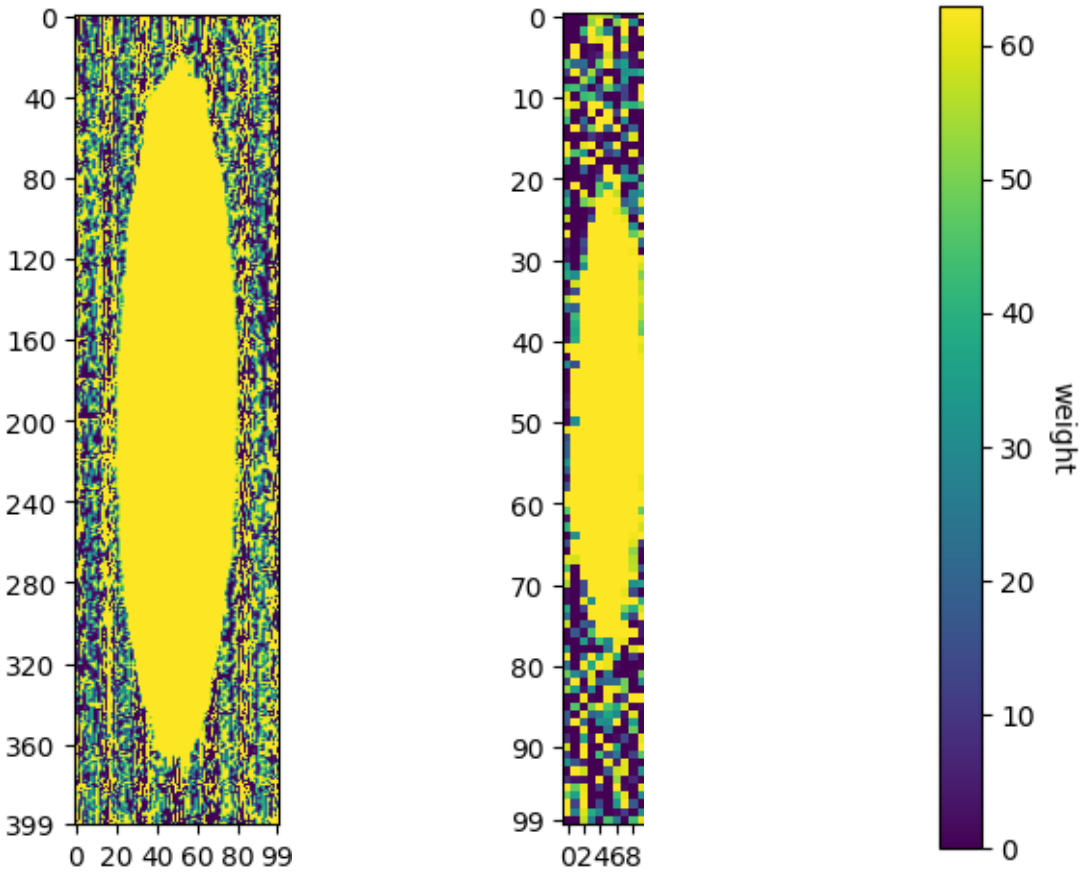


Figure 4.5: An illustration of a hidden unit rejecting a feature of a digit, thus avoiding the addition of high weights for output units other than the desired digit.

covered. This causes the network to identify digits as 6, instead of mostly 5, resulting in a higher accuracy due to it being able to identify the digit 6 as well as 5, resulting in an accuracy of 23%. At 30% the middle cluster impacts digit 5's column more and digit 5 is identified most often, with a few other digits occasionally identified, for an accuracy of 15%. This can be seen in Figure 4.8.

In Figure 4.8a at the 30% fault rate the column 5 corresponding to digit 5 is covered the most by the fault cluster. Thus at the 30% fault rate the digit 5 is mis-identified the most. At the 40% fault rate the middle cluster of faults expands to cover the columns around it's center slightly more evenly. Particularily, the column 6 corresponding to digit 6 is now covered slightly more evenly when compared to the column 5, as indicated by the red marking in Figure 4.8b. This is enough such that 6 is mis-identified slightly more frequently, but still not as much as the digit 5.

To show that the middle cluster's center which increases the weights of a column the most is indeed what makes the network misidentify 5 the most, an experiment is run where the middle cluster of stuck-at 1 faults is shifted towards the 6 digit's column. Doing so and proceeding to re-train the network with the 40% middle fault cluster shifted towards the 6 digit's column does result in the network misidentifying more digits as 6 rather than 5. The shifting of the cluster can be seen in Figure 4.9a and Figure 4.9b. The weights of



(a) Middle stuck-at 1 fault cluster example in the input to hidden layer.

(b) Middle stuck-at 1 fault cluster example in the hidden to output layer.

(c) Weight value mapping bar.

Figure 4.6: An example of a fault intolerant stuck-at 1 fault cluster case.

shifted cluster is highlighted in red in Figure 4.9b to show that the column 6 is now affected the most by the cluster. Figure 4.9a shows the original position of the cluster where 6 was not at the center of the cluster.

The 9 clusters placed in each of the 3 by 3 SRAM array sections case also suffers an accuracy drop due to the compounding effect of the fault clusters between the input to hidden layer and the hidden to output layer. This can be seen in Figure 4.10a. The hidden units around hidden units 18, 50, and 85 have a higher chance of activating and contributing to misidentifying digits with a stuck-at 1 cluster in their hidden to output

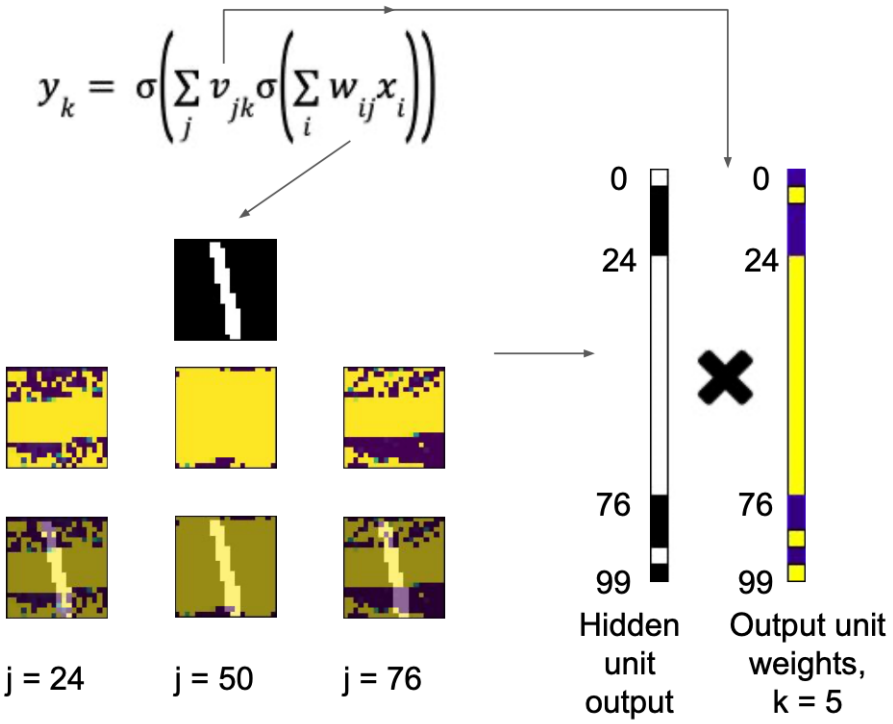


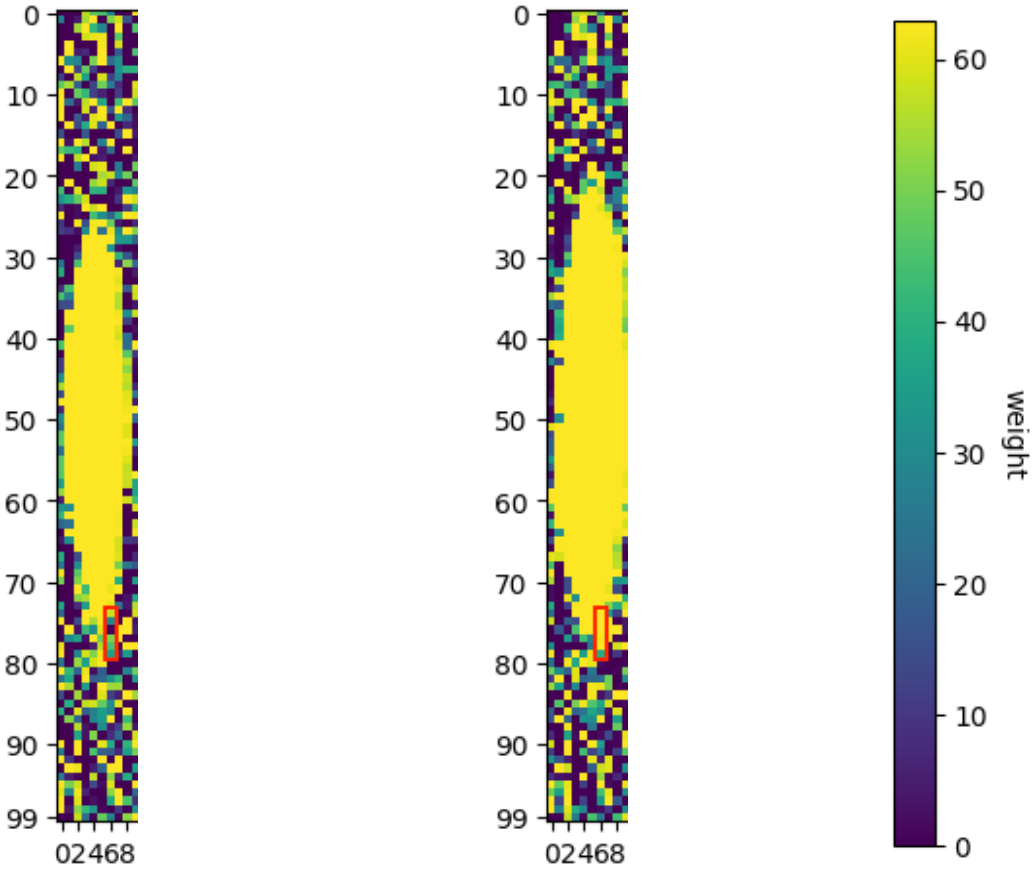
Figure 4.7: An example of calculating the output value for with the middle fault cluster.

connection weights. The faults in the hidden to output layer cause the network to guess 1, 5, and 8. These faults as can be seen in Figure 4.10b.

There is also an increase in accuracy moving from the 20% fault rate case to the 30% fault rate case. This is due to the more even distribution of faults at the 30% fault rate. The fault clusters in the hidden to output layer expand so that they more evenly cover the digit columns, spreading out the incorrectly identified digits. This can be seen in Figure 4.11b. This is as opposed to the 20% case which has a greatly increased chance of misidentifying digits as the particular digits the clusters are centered on, such as 1, 5, and 8.

4.3.3 Fault Tolerant Clustered Cases

Among the fault patterns, the clustered fault patterns which resulted in the highest accuracies were the top right and bottom left case. In these cases, the cluster in these fault



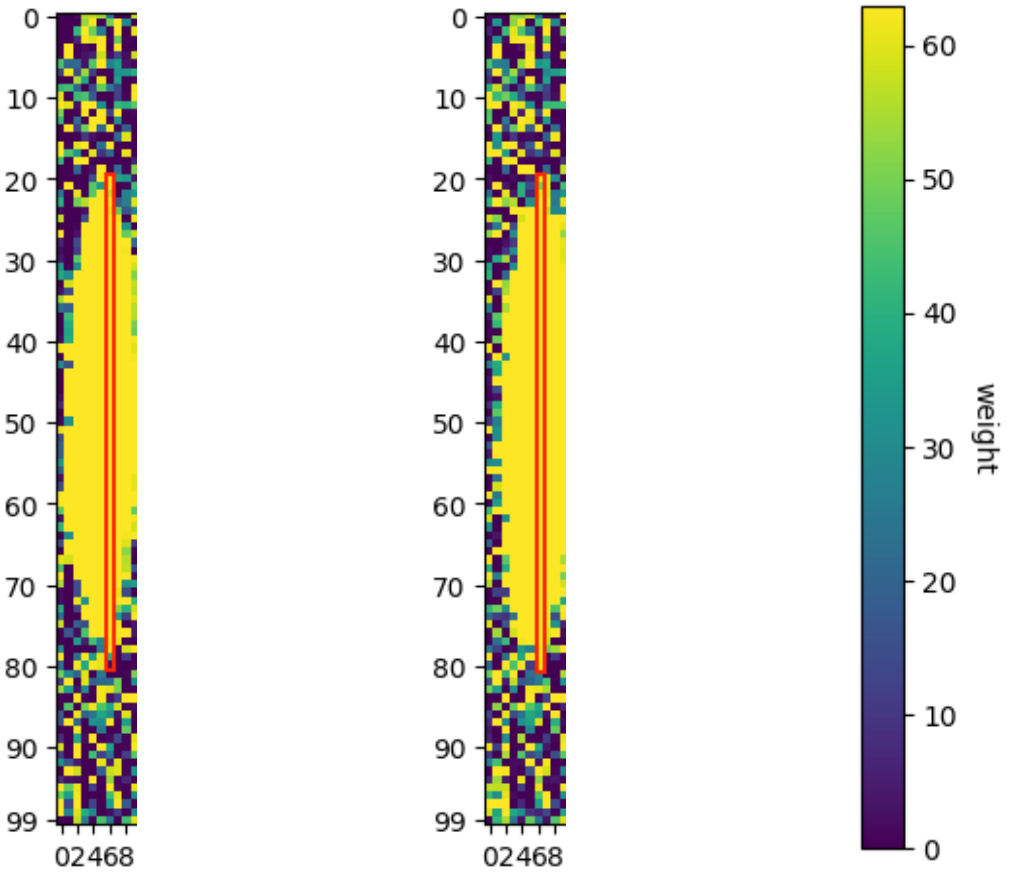
(a) The middle 30% stuck-at 1 fault cluster covering mostly the 5 column's weights in the hidden to output layer.

(b) The middle 40% stuck-at 1 fault cluster covering mostly the 5 column's weights and some more of the 6 column's weights.

(c) Weight value mapping bar.

Figure 4.8: The increase of the fault cluster size from 30% to 40% more evenly covers the 5 and 6 digit columns in the hidden to output layer.

patterns increase the weights of a section of hidden units, increasing their chance of activating. However, since the input hidden unit connection weights going from the hidden to output layer are not faulty in this case, the network can be trained with the appropriate weights to ignore any hidden unit activations which result from the clustered faults increasing their weights. This is illustrated in Figure 4.12. Figure 4.12a shows that the top right stuck-at 1 fault cluster affects hidden units 70 to 99, increasing their activation chance. Figure 4.12b shows that those same hidden units, 70 to 99, do not have a fault



(a) Original middle 40% stuck-at 1 fault cluster.

(b) Shifted middle 40% fault cluster covering mostly the 6 column's weights.

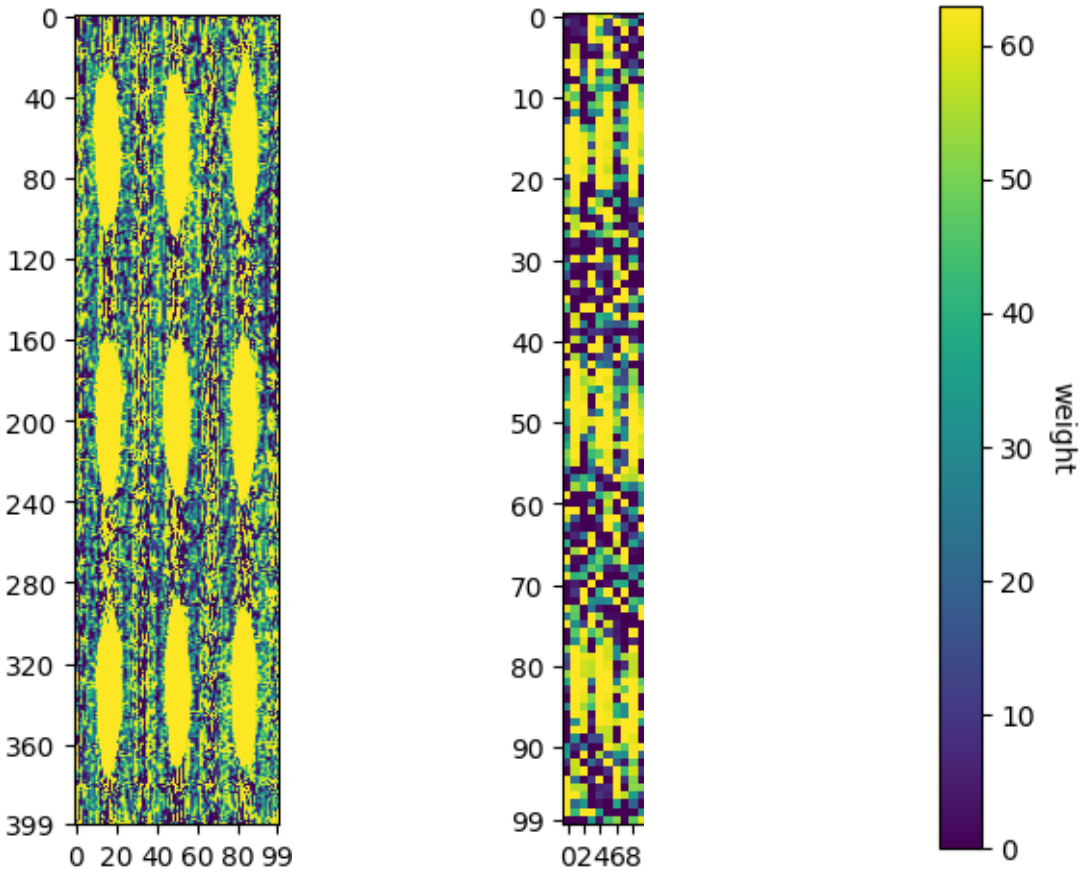
(c) Weight value mapping bar.

Figure 4.9: Shifting the 40% cluster to the 6 column causes the cluster to cover 6 more than 5's column.

cluster affecting them. Thus their weights can be adjusted in order to take into account the affect of the increased activation chance from the previous layer's hidden units.

4.3.4 Varying the Network Hidden Unit Size

With the effect of the random faults and clustered faults established the resilience of different network sizes is analyzed. As the input and output sizes are fixed to the sizes of



(a) Input to hidden layer weights with the 3 by 3 cluster fault pattern.

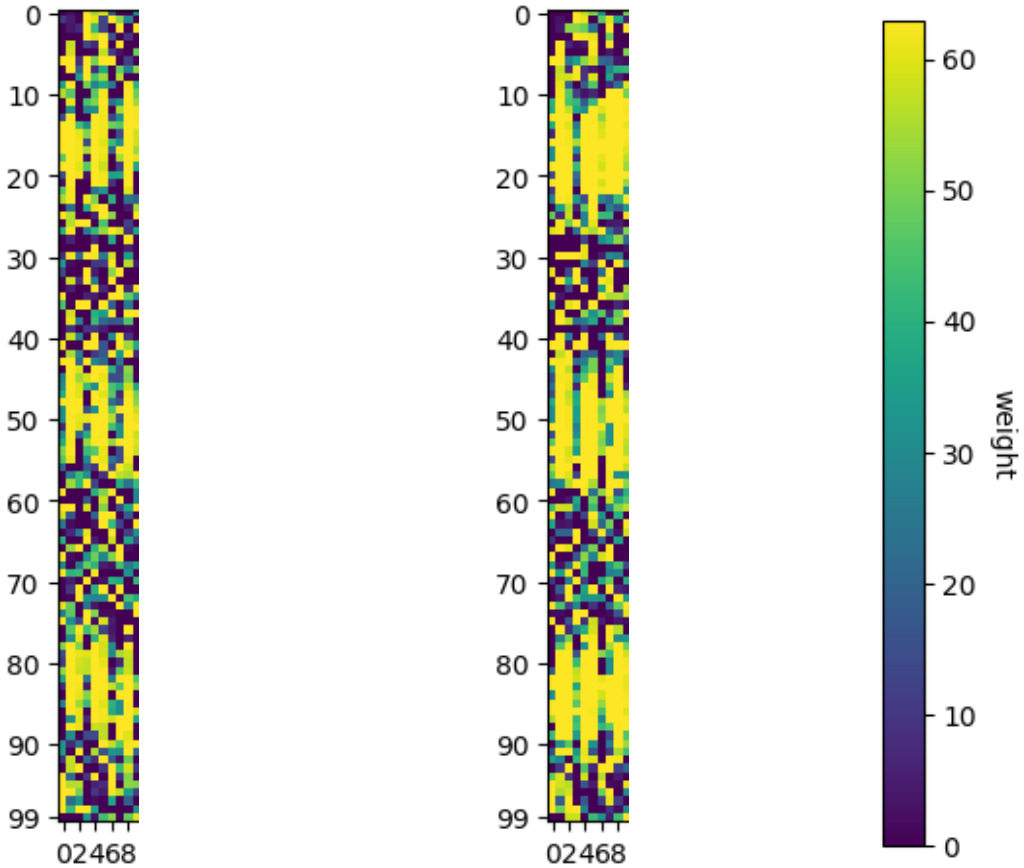
(b) Hidden to output layer weights with the 3 by 3 cluster fault pattern.

(c) Weight value mapping bar.

Figure 4.10: The network with 3 by 3 fault cluster patterns applied to both of it's SRAMs.

the 20 by 20 input images and the output fixed to the 10 classification targets, the size of the hidden layer is varied. The networks measured are by varying the number of hidden units by a factor of 2. The re-trained accuracy of networks with hidden units of size 50 and 200 are measured. The base accuracies of the networks with these hidden unit sizes is summarized in Table 4.1.

Figure 4.13 shows three plots for each of the three varied hidden unit sizes of 50, 100, and 200. The three plots cover the two general fault pattern cases, the random fault pattern and the clustered fault pattern. Under the clustered fault pattern case the most tolerable



(a) Hidden to output weights of the 20% 3 by 3 stuck-at 1 fault case.

(b) Hidden to output weights of the 30% 3 by 3 stuck-at 1 fault case.

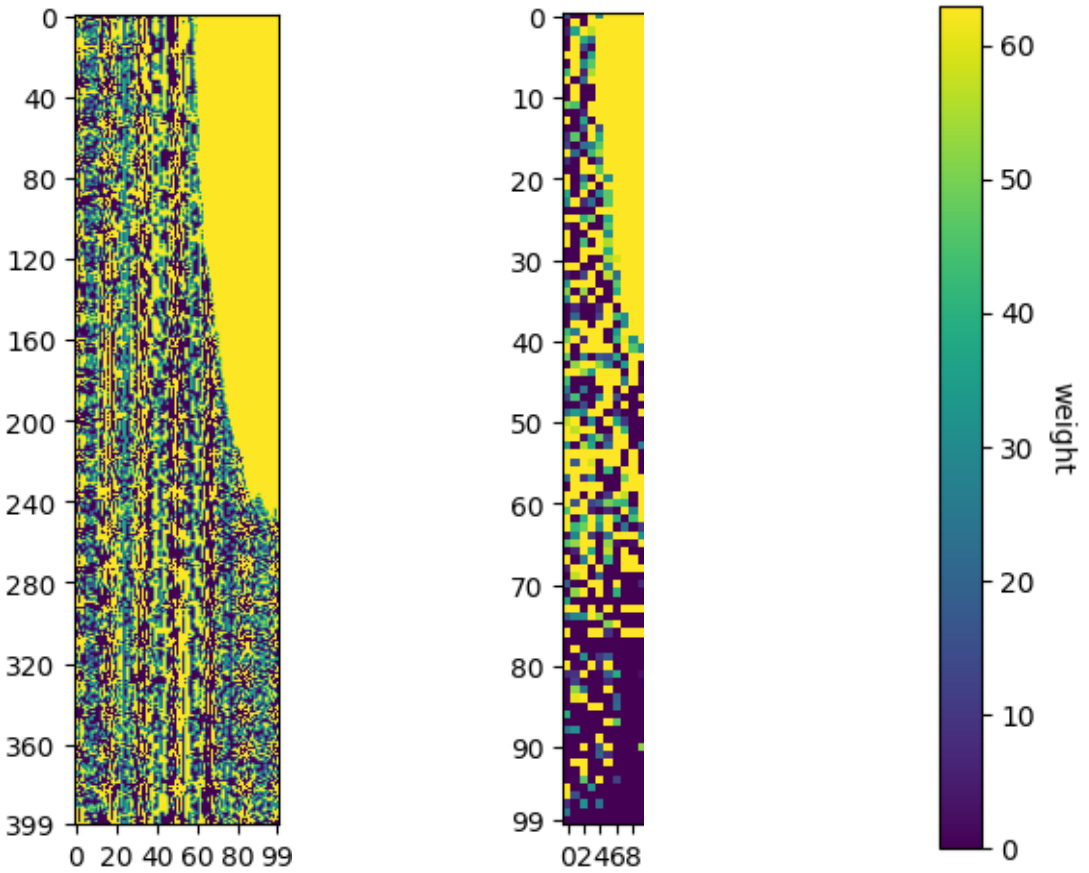
(c) Weight value mapping bar.

Figure 4.11: The increased width of the clusters at 30% more evenly distributes the faults, allowing the network to identify some more digits correctly.

Table 4.1: The base accuracies of the networks with different hidden unit sizes

Hidden Units	50	100	200
Base Accuracy	90.93	93.77	95.7

case and the least tolerable case are covered. These cases are the top right fault cluster and middle fault cluster pattern respectively. The 50 hidden unit case plots are marked in black, the 100 hidden unit cases are marked in blue, and the 200 hidden unit cases are marked in red. The middle fault cluster pattern cases are marked with a solid line, the



(a) The input to hidden layer weights trained with a top right cluster.

(b) The hidden to output layer weights trained with a top right cluster.

(c) Weight value mapping bar.

Figure 4.12: An example of a tolerable fault pattern with the top right cluster pattern.

top right fault cluster pattern cases are marked with a dashed line, and the random fault pattern caes are marked with a dotted line.

As Figure 4.13 illustrates, larger network sizes perform better than smaller network sizes. This can be seen as the 100 hidden unit cases performs better than the 50 hidden unit cases, and the 200 hidden unit cases perform better than the other two lower hidden unit cases. This shows that since the network has more partially usable weights and layers, it effectively has more usable hidden units. Thus larger networks perform better than smaller networks. The figure also shows that middle fault pattern cases are the least

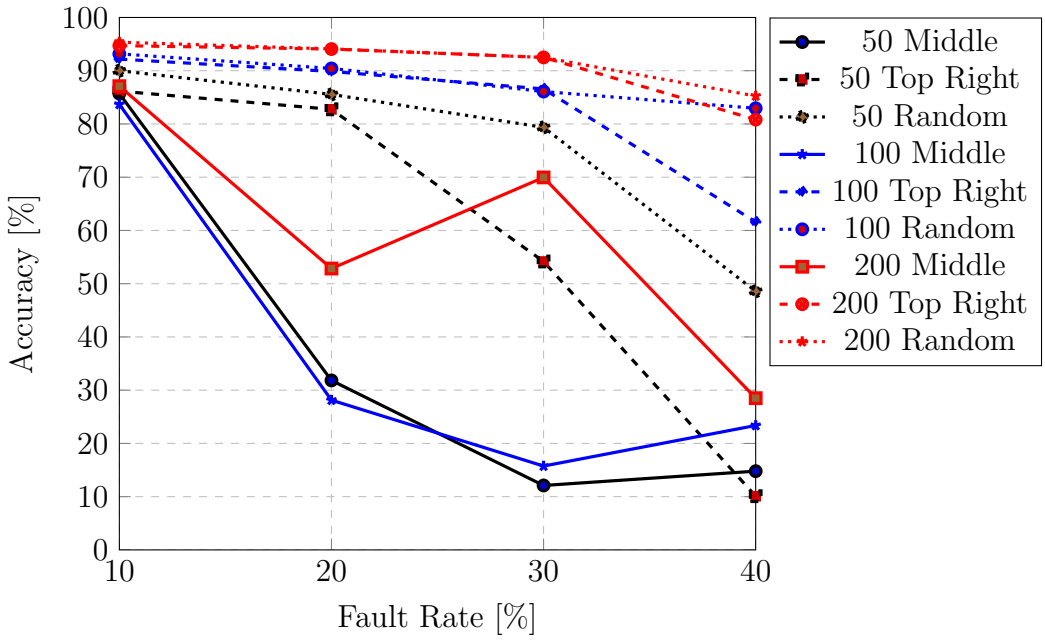
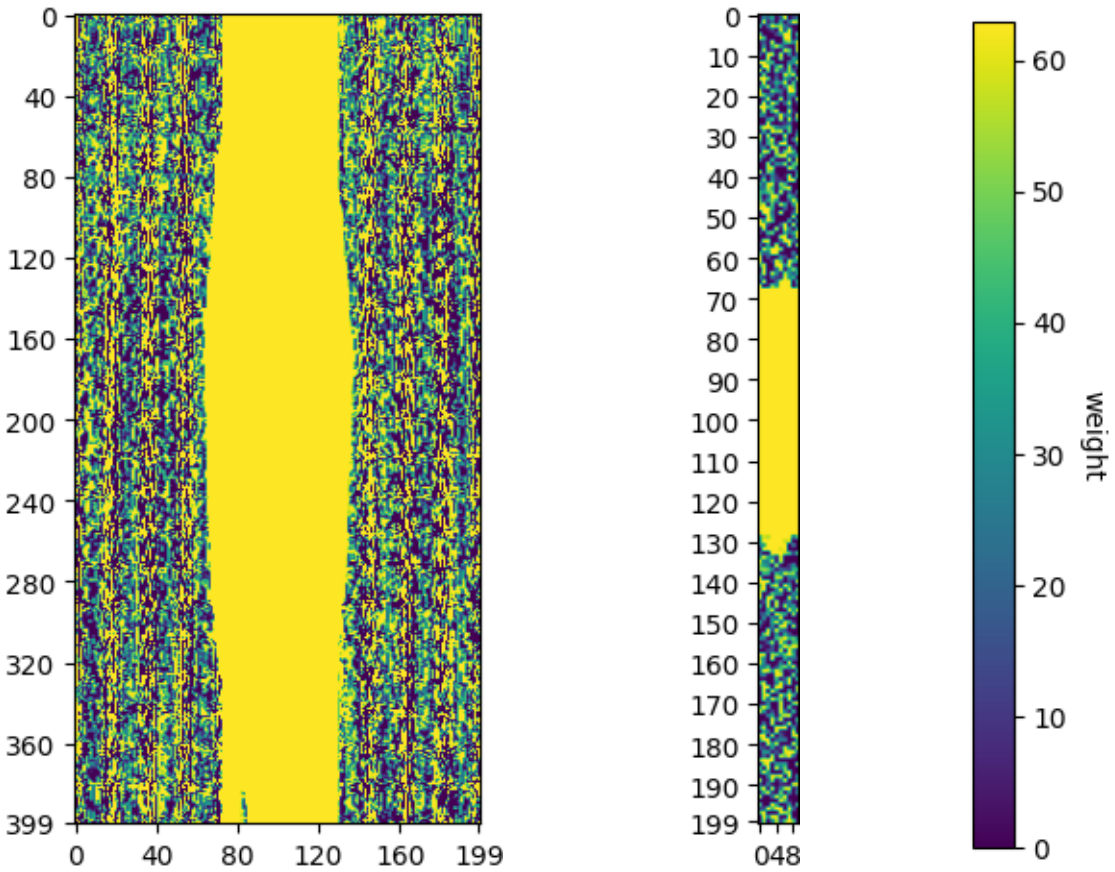


Figure 4.13: Accuracies re-training against various fault patterns with varied hidden unit sizes.

tolerable case, and that the random fault pattern is the most tolerable case. The top right fault cluster case performs between the two cases.

From Figure 4.13, one case of interest is the 200 hidden unit middle fault cluster case, where the accuracy for the 30% case rises compared to the 20% case. Similar to the 100 hidden unit case, the middle cluster expands and evens out, identifying and misidentifying the digits more evenly. To show that this is the case, an experiment is run where all columns which contain a stuck-at 1 fault have all of their bits set to 1 before re-training. Doing so improves the accuracy to 77.3%. Attempting the experiment without setting some of the columns at the edges of the cluster to all 1 improves the accuracy to 81.25%. Thus by evenly fixing the affected columns and rows in the input to hidden layer and hidden to output layer respectively, the effect of the stuck-at cluster is mitigated. The evenly applied stuck-at 1 faults mitigate the emphasis the stuck-at cluster applies to particular columns in the hidden to output layer. The evenly applied stuck-at 1 values in both network layers is illustrated in Figure 4.14. Figure 4.14a illustrates the stuck-at 1 faults being applied more evenly across the hidden units of the input to hidden layer. Figure 4.14b illustrates the stuck-at 1 faults being applied more evenly to the input hidden units of the hidden to

output layer.



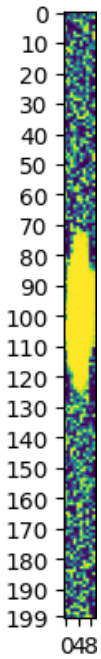
(a) The 200 middle fault case input to hidden layer. Faults are evenly applied across hidden units.

(b) The corresponding hidden to output layer.

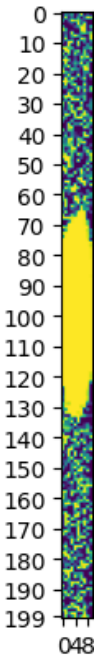
(c) Weight value mapping bar.

Figure 4.14: The 200 hidden unit network with stuck-at 1 faults applied to evenly distribute the faults across hidden units. This achieves an accuracy of 81.25%.

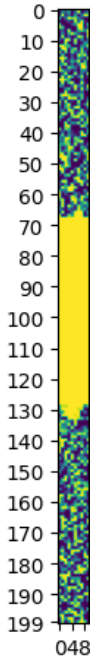
For clarity this is shown in Figure 4.15. Where in Figure 4.15a the middle cluster of faults unevenly affects the output units. They are more evenly affected at 30% as shown in Figure 4.15b and achieve a better accuracy. Finally, forcing the middle cluster faults to be more even as shown in Figure 4.15c results in the highest accuracy.



(a) Hidden to output layer at the 20% fault rate, the center stuck-at fault cluster is uneven at the center output units.



(b) Hidden to output layer at the 30% fault rate, the center stuck-at fault cluster is more even at the center output units.



(c) Forcing the bits around the center stuck-at fault cluster to stuck-at 1, forcing a more even stuck-at fault pattern. This achieves a higher accuracy.

Figure 4.15: The 200 hidden unit network with a middle stuck-at 1 fault cluster at different fault rates. A more even distribution of faults results in a better accuracy.

4.4 Stuck-At 0 Fault Pattern Resilience Results

In general, the re-trained stuck-at 0 fault pattern cases perform better. The fault patterns which perform worse are the top right and bottom left cases which previously performed better in the stuck-at 1 cases. The other cases performed just as well if not better than their stuck-at 1 counter part. This shows an interesting duality between the stuck-at 1 and stuck-at 0 clustered fault cases, where the stuck-at 0 case performs better where the stuck-at 1 performs poorly, and vice versa. This is illustrated in Figure 4.16. The accuracy of the stuck-at 0 cases are generally high and are able to tolerate high fault rates while only dropping about 3% in accuracy in the tolerable clustered fault cases or 11% at the

40% fault rate in the case of random faults similar to the stuck-at 1 case. The tolerable fault cluster cases are the middle, top left, bottom right, and 3 by 3 fault cluster cases.

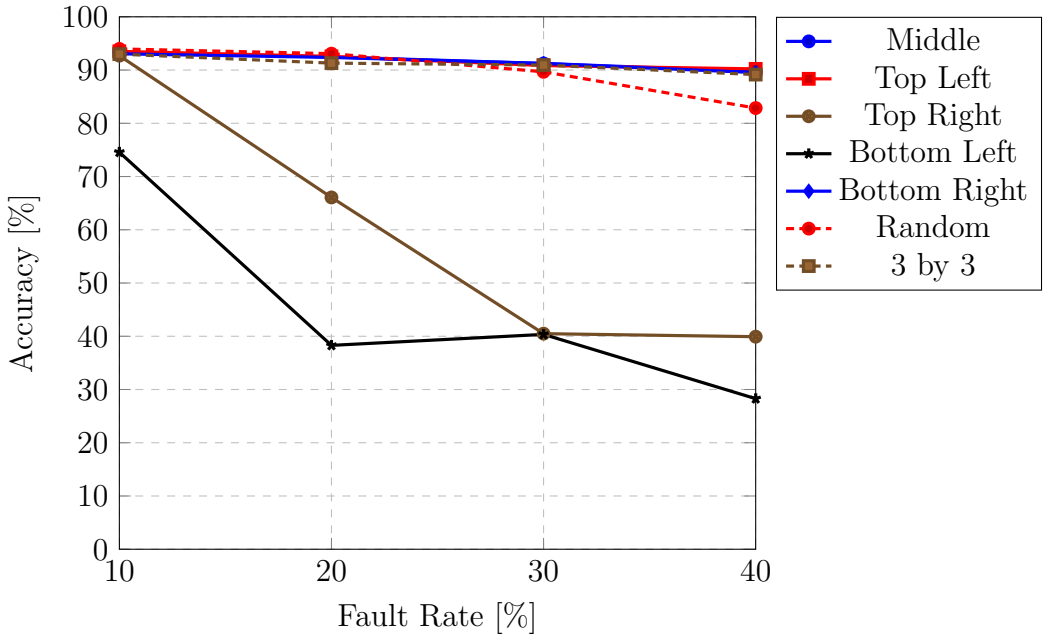


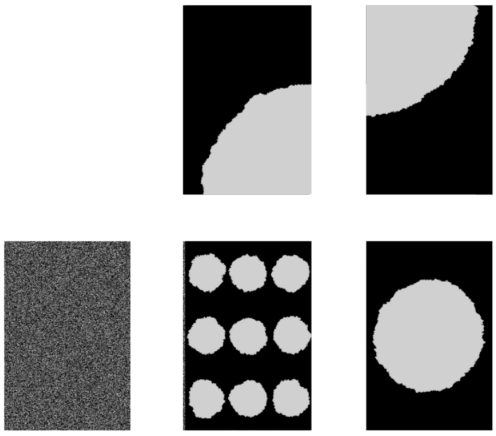
Figure 4.16: Accuracy after re-training the network under the middle, top left, top right, bottom left, bottom right, 3 by 3 cluster, and random fault patterns with stuck-at 0 faults.

For clarity, the fault pattern cases which are intolerable and tolerable are shown in Figure 4.17. Figure 4.17a illustrates the intolerable cases, and Figure 4.17b illustrates the tolerable cases.

4.4.1 Fault Tolerant Cases

As stated previously, the stuck-at 0 fault patterns which could be re-trained and perform well all are cases where their stuck-at 1 counter part performs poorly. Recall that in the stuck-at 1 case the faults compound between layers. In the stuck-at 0 case, since the faults set the weights to 0, the hidden units with many stuck-at 0 faults become less likely to activate. Because they are less likely to activate, and the hidden to output connection weights they compound with are already reduced in value due to the stuck-at 0 faults, this effectively results in slightly reducing the network size. Thus, as most of the weights can still be used, the network's accuracy does not suffer as much.

(a) The intolerable stuck-at 0 fault pattern cases.



(b) The tolerable stuck-at 0 fault pattern cases.

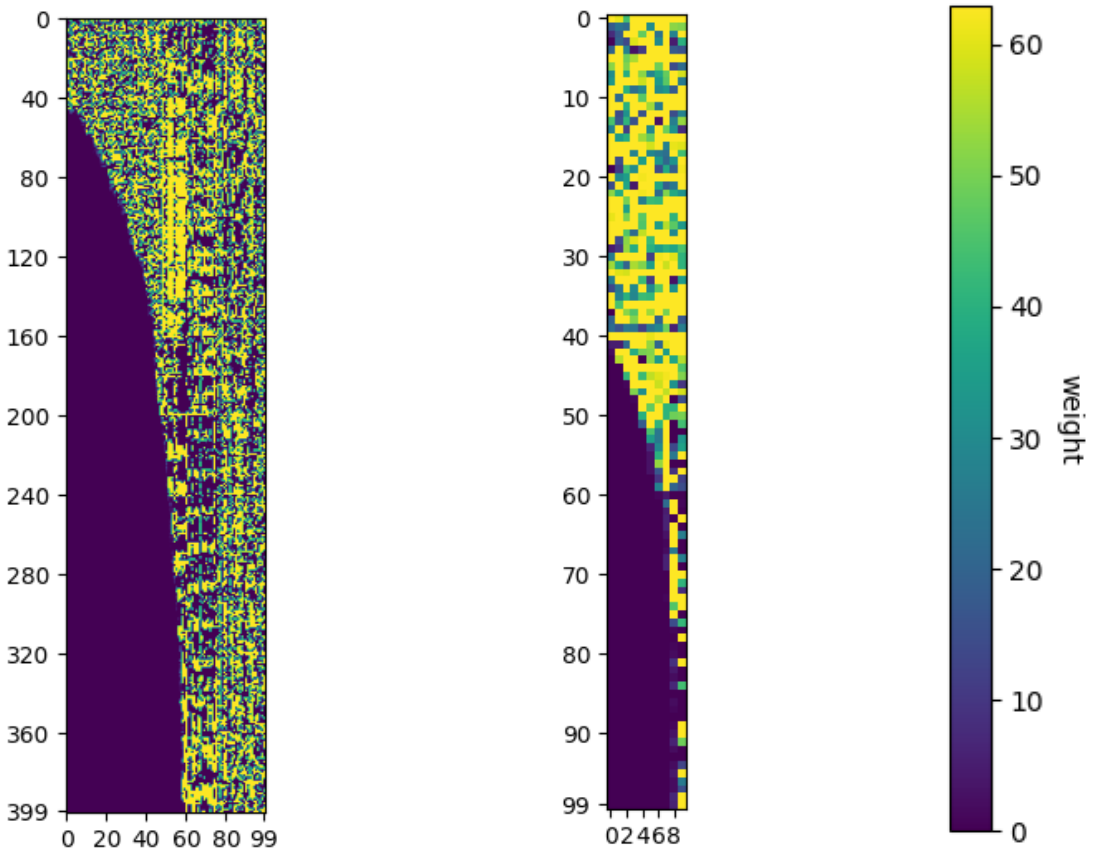
Figure 4.17: The tolerable and intolerable stuck-at 0 fault pattern cases.

4.4.2 Fault Intolerant Cases

The cases where the stuck-at 0 fault pattern cases perform worse than the stuck-at 1 faults are the top right and bottom left case. In these cases their stuck-at 1 counter parts performed better as the stuck-at 1 faults did not compound between layers. However the stuck-at 0 faults in the top right and bottom left cases render most of the network non-existent and unusable. This is as opposed to the other stuck-at 0 cases where the effect of the stuck-at 0 faults is minimized due to the fact that they overlap and affect both sides of the same small subset of hidden units from the two layers.

The effect of the intolerable stuck-at 0 fault pattern cases is illustrated by Figure 4.18 and 4.19. Figure 4.18a and 4.19 shows the stuck-at 0 faults reducing the weights and thus the chance of activating of hidden units 0 to 50. These hidden units which have a greatly reduced chance of activating are fed to the hidden to output layer's input 0 to 50. Shown particularly in Figure 4.19 with an example input digit of 1, most of the weights for the layers from 0 to 50 are greatly reduced, and thus are unlikely to activate. This produces the hidden unit output vector in Figure 4.19 where the output of hidden units 0 to 50 are 0, shown in black. The other weights from 50 to 99 can activate normally and produce 1's or 0's normally, where 1's are indicated with white. Thus these activation values of 0 from hidden units 0 to 50 effectively render the weights of the hidden to output layer corresponding to hidden units 0 to 50 unused. Additionally, the bottom left cluster

affecting the hidden to output unit effectively makes most weights in the 50 to 99 input range unusable as seen in Figure 4.18b and 4.19. In Figure 4.19, this is shown where the weights corresponding to hidden units 50 to 99 are greatly lowered in the hidden to output layer for lower output units 0 to 7. The output unit 9 has the most usable weights left over that can be used with hidden units 0 to 50 as shown in Figure 4.18b. As a result the digit 9 is mis-identifed the most. Thus this illustrates how most of the network becomes unused with the top right and bottom left stuck-at 0 fault cluster patterns.



(a) The input to hidden unit layer of the top right stuck-at 0 case.

(b) The hidden to output unit layer of the top right stuck-at 0 case.

(c) Weight value mapping bar.

Figure 4.18: An example of a intolerable stuck-at 0 fault pattern case, in particular the bottom left cluster pattern.

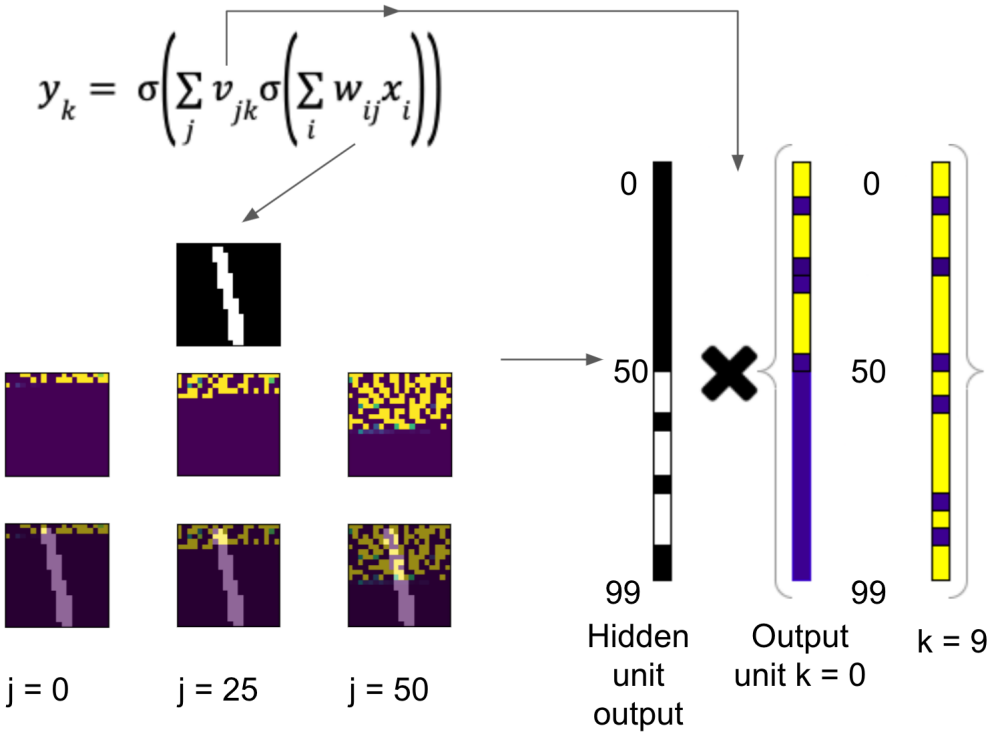


Figure 4.19: Example of calculating the output values in the bottom left case.

4.5 Stuck-at 1 and 0 Fault Differences Summary

Between the stuck-at 1 fault and stuck-at 0 faults, stuck-at 0 faults are more tolerable than stuck-at 1 faults. Of the 6 fault patterns under analysis, stuck-at 1 faults cause 3 cases to have greatly lowered re-trainable accuracy. These fault cases are clustered fault cases. This is due to the compounding effect of the faults between the layers of the network, where the stuck-at 1 faults increase the activation chance of hidden units which are connected to weights in the next layer that also suffer from stuck-at 1 faults. Additionally, in general the stuck-at 1 faults result in a lower accuracy. Conversely, the stuck-at 0 faults generally result in a higher re-training accuracy. Interestingly the cases where the stuck-at 0 faults perform worse than the stuck-at 1 faults are the cases where the stuck-at 1 faults performed the best. These cases have the stuck-at 1 fault clusters positioned to increase the activation chance of hidden units which do not lead to weights with stuck-at 1 faults in the successive layer. However with stuck-at 0 fault clusters, the fact that these stuck-at 0 fault clusters are spread around mean that they greatly reduce the hidden unit's chance of activation.

This effectively greatly reduces the size of the usable network. Table 4.2 summarizes the tolerable and intolerable clustered and random fault patterns between the stuck-at 1 and stuck-at 0 cases.

Table 4.2: A summary of tolerable and intolerable fault pattern cases with stuck-at 1 and stuck-at 0 faults.

	Middle	Top L.	Top R.	Bottom L.	Bottom R.	Rand.	3 by 3 Clusters
S.A. 1	Intol.	Intol.	Tol.	Tol.	Intol.	Tol.	Intol.
S.A. 0	Tol.	Tol.	Intol.	Intol.	Tol.	Tol.	Tol.

Chapter 5

Fault Recovery

5.1 Inverted SRAM Array Accesses

One simple fault recovery scheme to overcome the effect of the position of the clustered fault pattern cases is to alter the accesses to the SRAM array. The simplest way to achieve this is by inverting or flipping the accesses to the SRAM rows and columns along the middle in each direction. Doing so would allow a fault intolerable case, such as the top left stuck-at 1 fault cluster case, to flip the accesses to either the input to hidden or hidden to output layer SRAM such that the faults do not compound between the layers. This reduces the fault pattern case to one of the tolerable cases which for example are the top right and bottom left in the case of the stuck-at 1 faults.

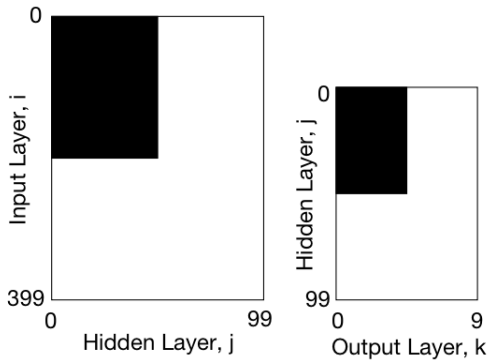
Figure 5.1 provides an example of that this would look like for the top left fault cluster case. Figure 5.1a shows the original SRAM arrays with the faults marked in black at the top left for the input to hidden layer and hidden to output layer on the left and right respectively. Figure 5.1b shows the result of inverting the row accesses to the hidden to output SRAM array. Although the faults are located in the top left corner, the read accesses now start reading from the bottom of the array. Thus the faults which were originally at the top left of the array now appear to be at the bottom left of the array, and the faults do not compound between layers. Using the bottom left stuck-at 0 case and flipping the row accesses of the hidden to output layer results in similar accuracies to the tolerable stuck-at 0 cases. This is shown in Table 5.1.

This fault recovery technique requires 2 bit per layer to mark whether or not the row or column accesses are inverted. Some addition hardware is required to route the row and

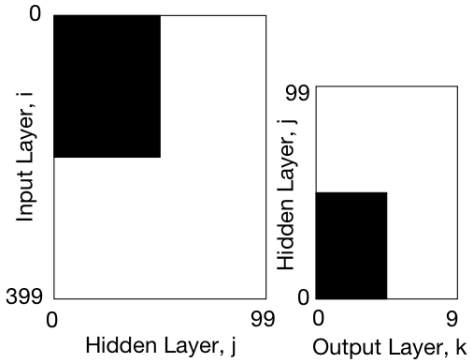
Table 5.1: Accuracy of the base stuck-at 0 bottom left case and after with the hidden to output layer's row access inverted

Fault Rate	10%	20%	30%	40%
Bottom Left	74.57%	38.29%	40.35%	28.28%
With Row Inversion	93.24%	92.4%	91.17%	89.84%

column access accordingly. This could be done simply with de-multiplexers between the inverted and non-inverted rows. This would not work however on fault patterns which cover both sides of the arrays equally, such as the middle fault cluster cases.



(a) An example of the top left stuck-at 1 fault cluster pattern case.



(b) An example with the row accesses to the hidden to output SRAM flipped.

Figure 5.1: An illustration of the effect of flipping the SRAM array accesses to the hidden to output SRAM. Black represents stuck-at 1 faults. White represents functioning SRAM cells.

5.2 Distributed Weight Bits

In order to improve the worst fault pattern case, the middle fault cluster case, the effect of the centered fault cluster must be mitigated. To mitigate the effect of the center cluster of faults, one possible approach is to disperse the effect of the stuck-at 1 faults by distributing the bit placement of the weights such that the center cluster of faults is spread out across multiple weights, reducing it's overall effect in individual weights. This can be achieved by storing the first MSB of the weights in the SRAM arrays first, then the second MSB, and so on. Figure 5.2 presents an example of a distributed data bit storage scheme with two 3

bit data entries. Figure 5.2a shows how a standard data bit storage method would store all three bits of one data entry contiguously before storing the next data entry. The MSB, or bit 2, is marked in red, the second MSB, or bit 1, is marked in green, and the third MSB or Least Significant Bit (LSB) is marked in blue. The subscript marks which data entry the bit belongs to, either entry 0 or entry 1 in this example. Figure 5.2b illustrates how the distributed data bit storage method would first store the MSBs of the data entries 0 and 1 first, as marked in red, and then subsequently the second MSB before finally storing the third MSB or LSB.

(a) A standard data bit storage method.

(b) A distributed data bit storage method.

Figure 5.2: An example of distributing bits of data in a non-contiguous manner.

5.2.1 NeuroSim Changes

The changes to the NeuroSim C++ code required to simulate training runs with this bit distribution are within the variation object code. The variation object's helper functions which apply faults of to a weight of a particular column row index is modified as it applies the faults to a provided weight. In particular, the helper function which constructs the faulty value that is combined with the weight is modified to read the fault enumeration values in a distributed manner across the SRAM array row.

5.2.2 Middle Cluster Fault Pattern Case

Re-training the network with these distributed weights against the middle fault cluster case results in a large accuracy improvement from the original case. This can be seen in Figure 5.3.

Observing the weights of the trained network, as the middle cluster mostly lands on the lower bits of the weight as the MSBs are located towards the edge of the SRAM array, the weights are increased to at most half the maximum weight value. At half the maximum weight value, the contribution of the weight towards the sigmoid activation function if it's used is approximately 0 since the minimum is mapped to -1.0 and the maximum is mapped

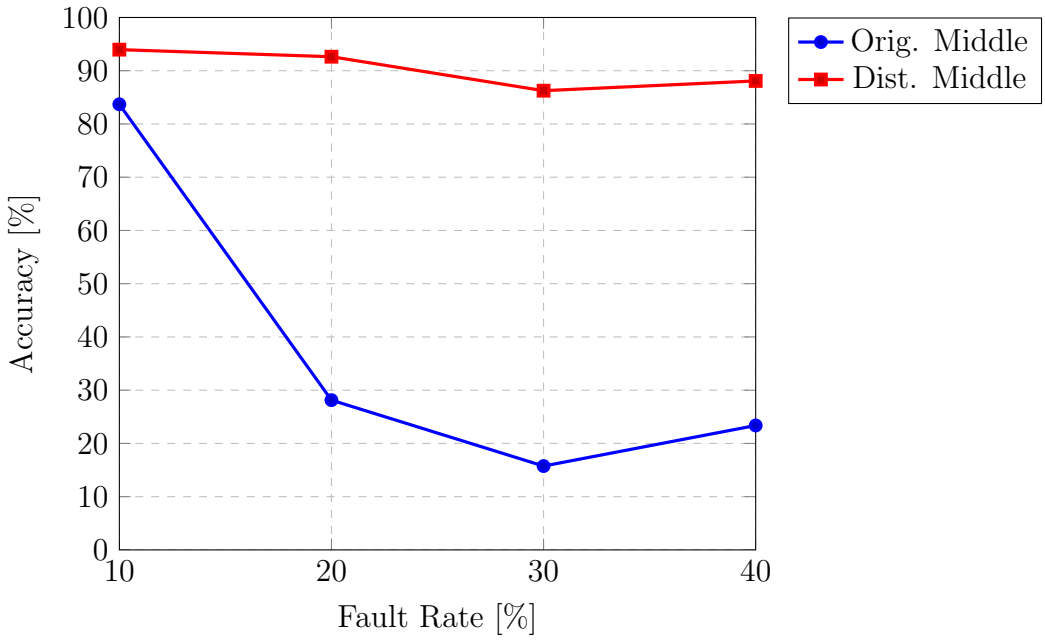
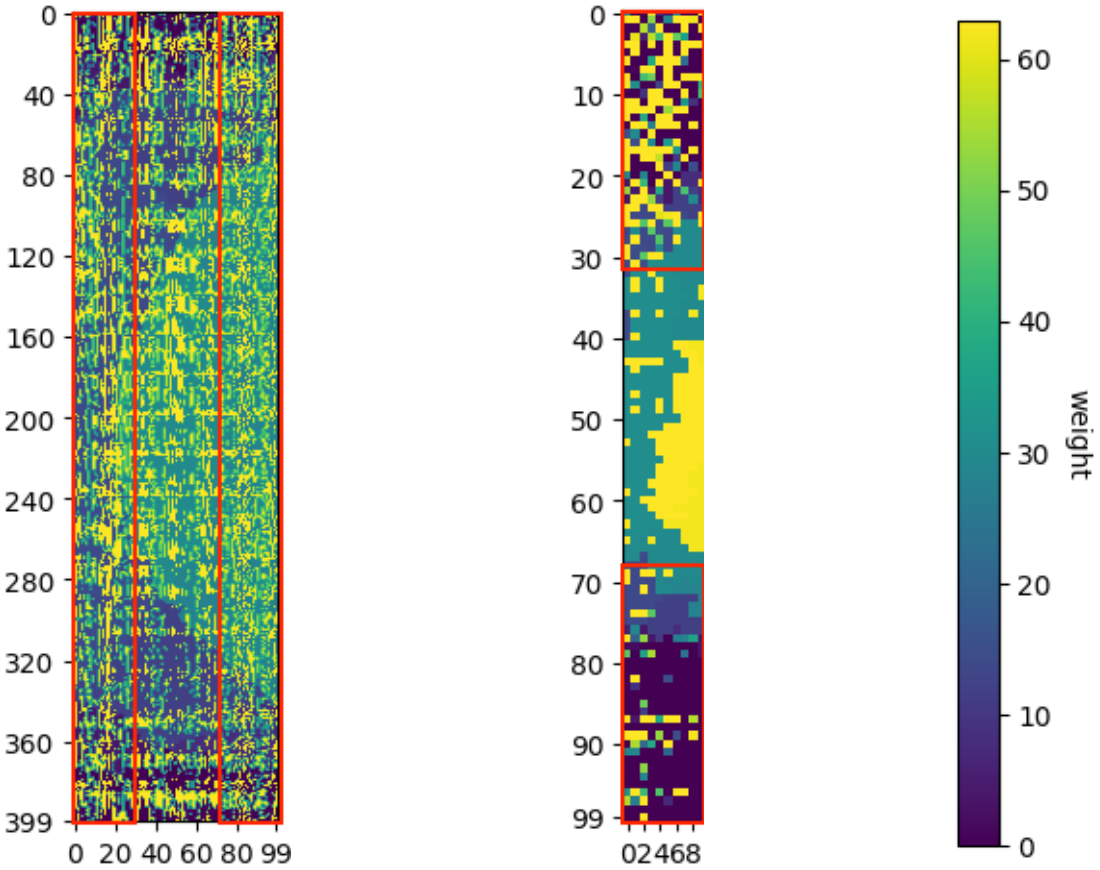


Figure 5.3: Accuracy comparison between the original middle stuck-at 1 fault cluster case and the distributed weight bit version of the case.

to 1.0 for usage with the sigmoid function. This effectively reduces the impact of the stuck-at 1 faults. Figure 5.4 illustrates that the network is still usable at 40%. The input to hidden layer is wide enough such that the middle cluster doesn't land on the MSBs of the layer, as seen in Figure 5.4a. The hidden to output layer is smaller width-wise as it has 10 columns for the 10 output digit values. Thus due to it's greater aspect ratio the fault cluster lands on some of the weight's MSBs, increasing some of the weights corresponding to hidden units 40 to 70 as seen in Figure 5.4b. Even with the center weights stuck-at 1 in the hidden to output layer the weights at hidden units 0 to 30 and 70 to 99 are still usable, as marked in red. As such the input to hidden layer weights can be trained to make use of these hidden to output layer weights.

5.2.3 Top Left Cluster Fault Pattern Case

As the accuracy improves due to the placement of the middle cluster of faults no longer falling on the MSBs, it is pertinent to examine a case where the cluster of faults lands on the MSBs. It is expected that by covering the MSBs, the recovery scheme will not be able



(a) Input to hidden layer with distributed weight bits.

(b) Hidden to output layer with distributed weight bits.

(c) Weight value mapping bar.

Figure 5.4: The distributed weight bit technique applied to the middle cluster of stuck-at 1 fault case.

to improve the accuracy. Thus a case where the faults lie on the MSBs, the top left fault cluster case, is considered. As illustrated in Figure 5.5 the accuracy cannot be recovered at higher fault rates of 30% to 40%, but offers some improvement at the 20% fault rate.

These stuck-at faults at higher fault rates affect most of the weights' MSBs and result networks with an accuracy of 10%, the accuracy of only guessing one digit across the input images. At the 20% fault rate the accuracy is improved since the faults are distributed somewhat more evenly across the upper rows in the network as opposed to only affecting a few columns in a corner. This is illustrated in Figure 5.6. In Figure 5.6b weights

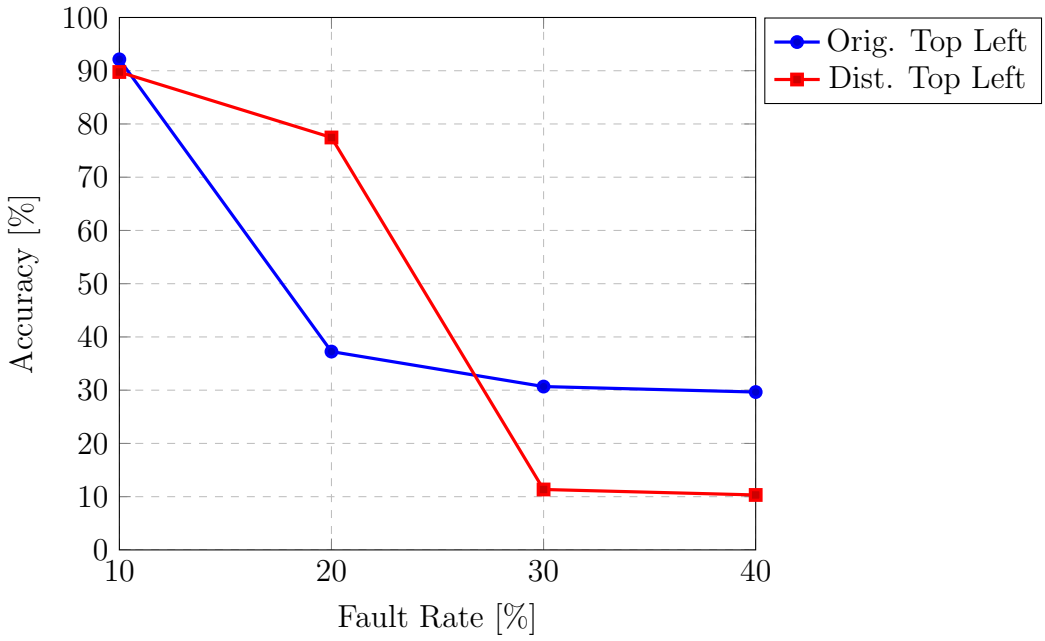
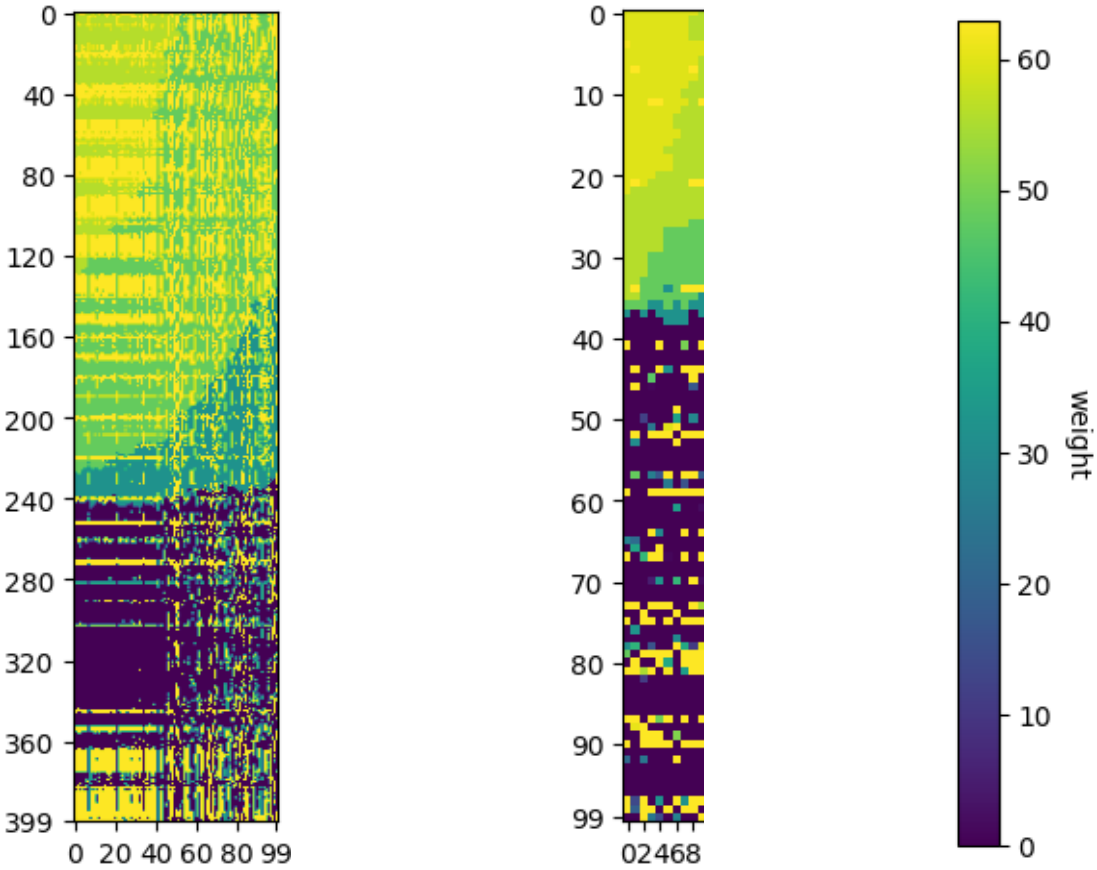


Figure 5.5: Accuracy comparison between the original top left stuck-at 1 fault cluster case and the distributed weight bit version of the case.

corresponding to hidden units 0 to 40 are mostly stuck at high values, but the remaining weights can still be used. While the input to hidden layer, shown in Figure 5.6a is mostly covered in faults in the the upper portions of the hidden unit columns, the columns can still be used, albeit to a lower degree. Additionally, as the fault cluster is centered around the top left corner of the SRAM array, the weights in the top left corner are affected more. Moving away from the top left corner, the the weights have fewer faults and the hidden units closer to the right side of the array are more usable than the ones on the left side.

The network still suffers a drop in accuracy at the 20% stuck-at 1 fault rate due to the stuck-at 1 faults obscuring part of the upper section of the hidden units. While the network is able to detect digit features with a large portion of it's weights set to higher values in the upper portion of the hidden units, this still obscures and interferes with the feature detection ability of the hidden units. This results in some digits such as 5 or 9 activating some hidden units which are intended for the digit 7 as seen in Figure 5.7. Figure 5.7b shows how a hidden unit can still be used to detect features for the digit 7. Figure 5.7a shows how the upper portion of the network obscures the differences between some digits, or in this case the digits 9 and 7. Thus the hidden unit intended for detecting a feature of



(a) Input to hidden layer with distributed weight bits.

(b) Hidden to output layer with distributed weight bits.

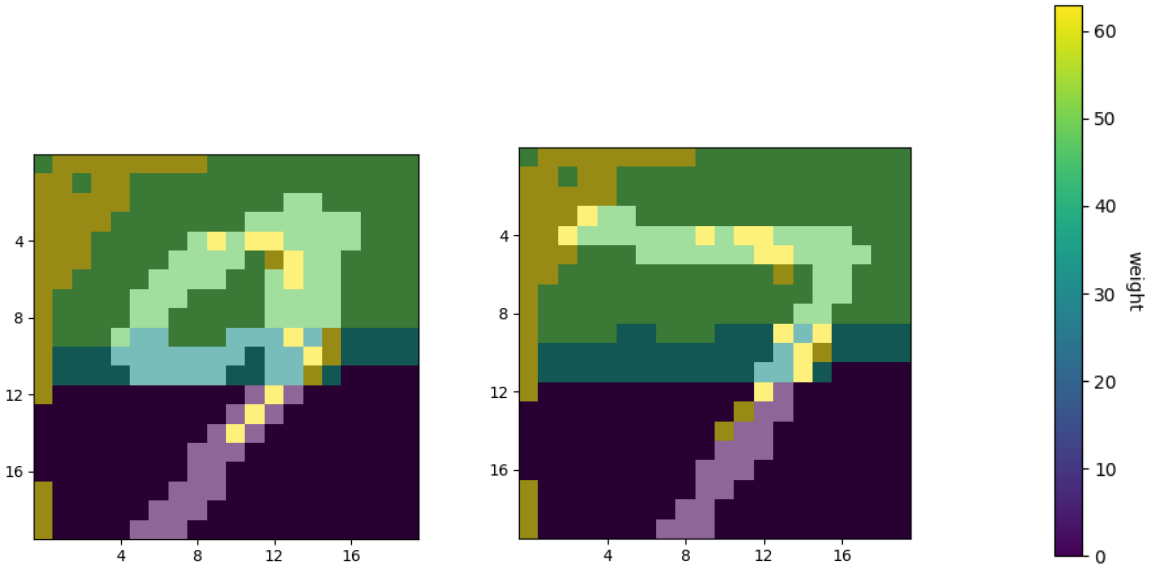
(c) Weight value mapping bar.

Figure 5.6: The distributed weight bit technique applied to the top left fault pattern case.

the digit 7 can be also activate when presented with the digit 9, resulting in the network mis-identifying the 9 as 7.

5.2.4 Random Fault Pattern Case

After measuring the resilience of the network with the clustered fault patterns, the random fault case is now considered. Training the network against random faults with the distributed bit weights results in similar accuracies to the original random stuck-at 1 fault case as seen in Figure 5.8. This is expected as the random fault case is intended to be a



(a) The digit 9 activating a hidden unit intended for the digit 7.

(b) The digit 7 activating a hidden unit used to detect features of the digit 7.

(c) Weight value mapping bar.

Figure 5.7: An example of the distributed weight bit recovery technique obscuring the upper portions of some digits.

uniformly distributed random fault case, and should affect the distributed bit weight case in a similar manner.

5.2.5 Varying the Bit Precision to 5, 6, and 8 Bit

Beyond the fault pattern cases, the resilience of the network is tested against various bit precisions. The variety of bit precisions that are measured are 8 bit and 5 bit, two bits more and one bit less than the nominal 6 bit case. Selecting the 5 bit case is a result of the 4 bit case lacking enough precision to be used as it is unable to be trained beyond an accuracy of 10%, the accuracy of the network selecting and misidentifying one digit regardless of the input. The base accuracies of the networks are shown in Table 5.2. The base 6 bit and 8 bit precisions have a similar base accuracy, and the 5 bit precision has an accuracy about 1.3% lower than the higher bit precision cases due to it's lower precision.

From Figure 5.9 the effect of varying the weight precision to 8 bits resulted in an accuracy similar to the base 6 bit case. The wider SRAM array in the 8 bit case may

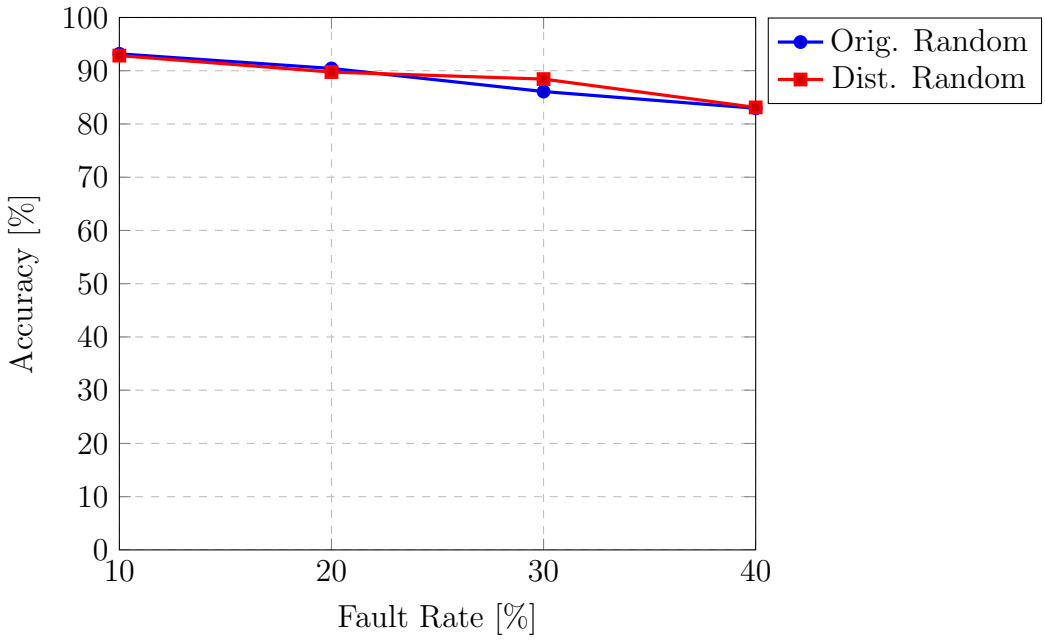


Figure 5.8: Accuracy comparison between the original random stuck-at 1 fault cluster case and the distributed weight bit version of the case.

Table 5.2: The base accuracies of the networks with different precisions for the weight bits.

Precision	5 Bit	6 Bit	8 Bit
Base Accuracy	92.44	93.77	93.62

smooth out the re-trained accuracies attained at each fault rate but does not provide any other major accuracy improvements. The 5 bit precision when compared to the base 6 bit case performs poorly at the 40% fault rate. This is attributed to the SRAM array shrinking in width and increasing the aspect ratio due to the fewer bits used. This causes the middle cluster of faults to reach the sides of the array, in this case the left side, where the MSBs are stored.

Figure 5.10 shows that between the 5, 6, and 8 bit precisions used, the effect of the random stuck-at faults are the approximately the same. This result is to be expected as the faults are uniformly randomly distributed, and the same percentage of bits are faulty at each fault rate. Thus the effect should be approximately the same in each case.

With the top left fault cluster applied and the bit precision varied, the 8 bit case

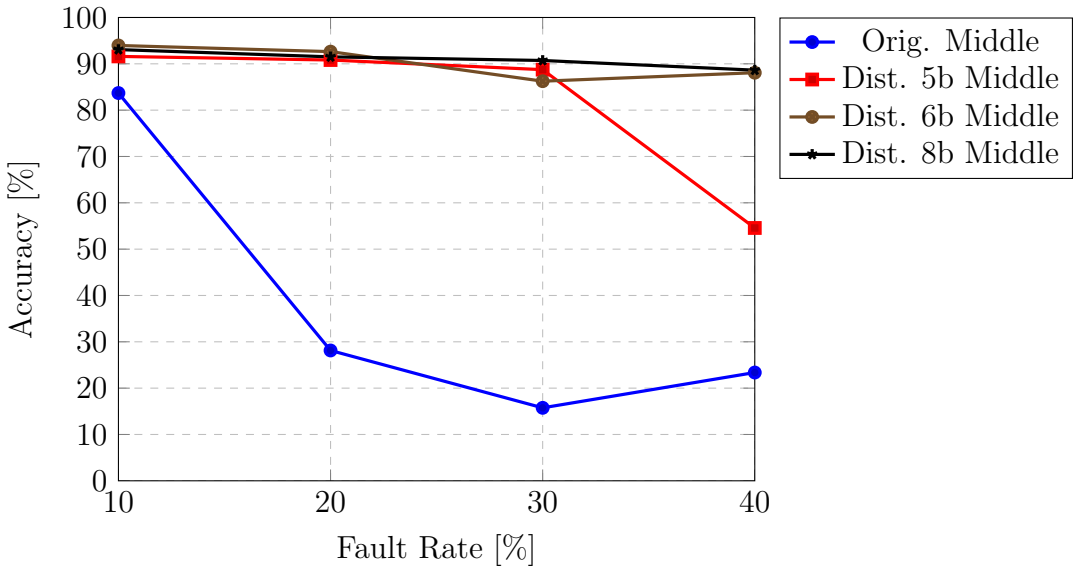


Figure 5.9: Accuracy comparison between the original middle stuck-at 1 fault cluster case and the distributed weight bit version of the case at varying bit precisions.

tolerates the top left cluster less than the base 6 bit case and the 5 bit case performs slightly better, if not approximately the same as the base case. Figure 5.11 shows that the 8 bit case no longer is able to partially recover some accuracy at the 20% fault rate compared to the base case. The extra precision of the 8 bit case increases the maximum value the network and makes it harder to adjust the weights to account for these greatly increased values. Conversely, the 5 bit precision does not suffer this issue. Instead it maintains an accuracy similar to the base 6 bit case, with a marginal improvement at 20%.

5.3 Weight MSB Protection

As the distributed weight bit case indicates in the middle fault cluster and the top left fault cluster cases, the integrity of the weights' MSB is critical. Thus a fault recovery method which protects the MSB of the weights is evaluated.

In order to set a baseline for the expected accuracy of a weight MSB protection technique, a re-training run is performed where any faults in the MSB of the weights are ignored. This is run with the stuck-at 1 middle cluster case to measure this technique's potential to overcome the worst case.

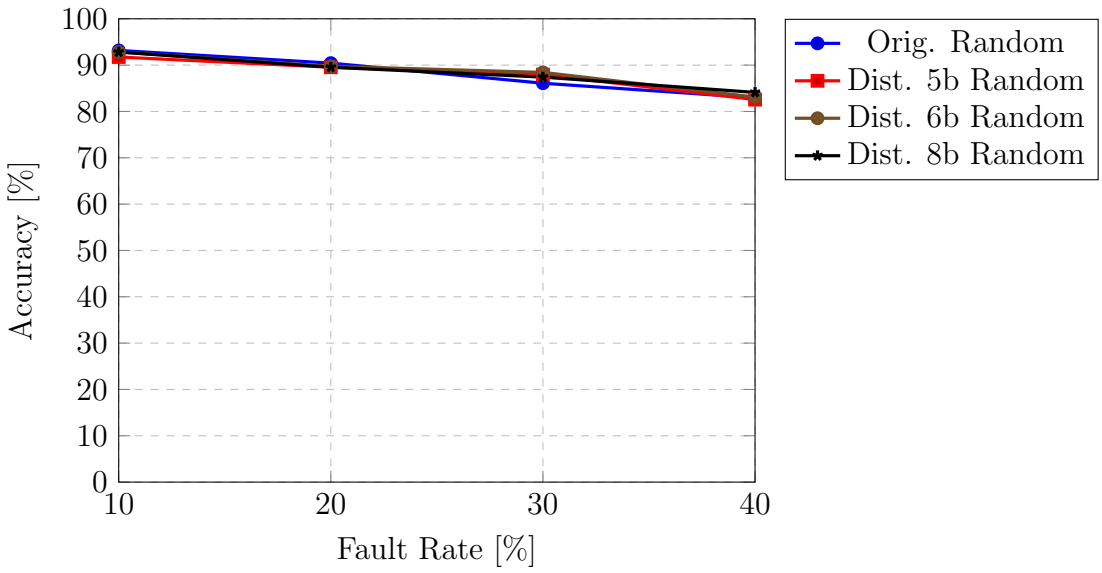


Figure 5.10: Accuracy comparison between the original random stuck-at 1 fault cluster case and the distributed weight bit version of the case at varying bit precisions.

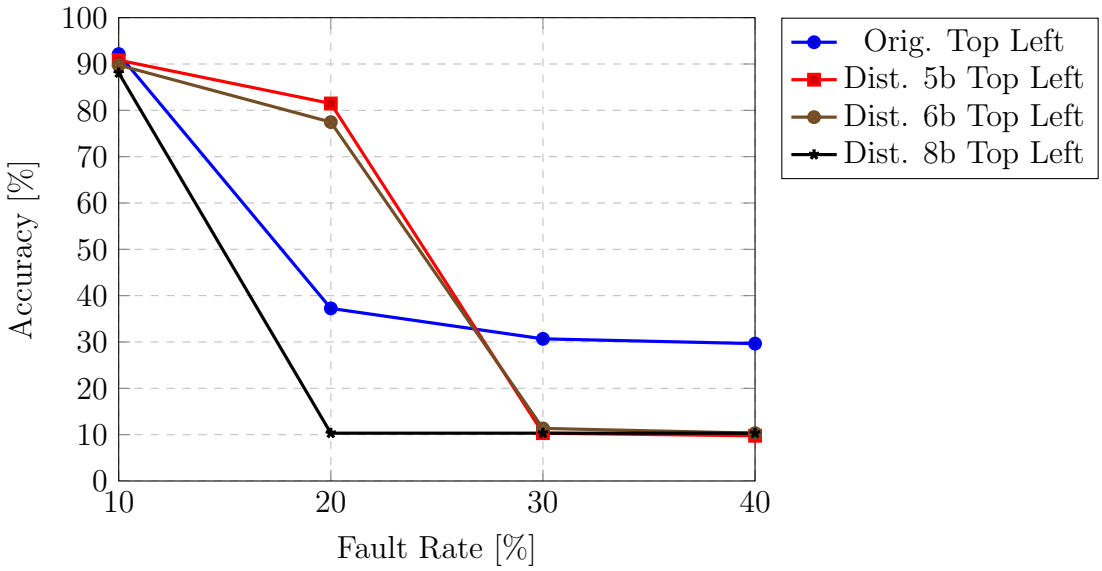


Figure 5.11: Accuracy comparison between the original top-left stuck-at 1 fault cluster case and the distributed weight bit version of the case at varying bit precisions.

To implement a MSB protection scheme, a (3,1) Hamming code is used, where 2 parity bits are used to protect the MSB. In the (3,1) Hamming code when the bits are read out, the number of 1's and 0's are compared and the one which appears the most is used. As an example, if there are two 1's and one 0, then the value 1 is used. This scheme is thus able to protect the MSB if there is one or less fault where the fault sets a bit to the opposite of the intended value in the 2 parity bits and 1 MSB. Two or more of such faults is unrecoverable. As we are measuring stuck-at 1 case, this means that two or more faults fixes the weight's MSB and thus it's value to at least half the max value. The resulting storage overhead of the MSB protection scheme for the base 6 bit weight case is thus 25%. In other terms, the overhead is 2 parity bits per each 8 bits used to represent a weight in total. Figure 5.12 shows an example of a 6 bit weight and a 6 bit weight with 2 parity bits.

(a) A base 6 bit weight.

(b) A 6 bit weight and 2 parity bits.

Figure 5.12: An example showing a base 6 bit weight and a 6 bit weight plus 2 parity bits.

5.3.1 NeuroSim Changes

The changes in NeuroSim required to represent this MSB protection scheme are mainly in the variation object's weight read out helper function. The variation object is first set to 2 plus the number of weight precision bits so the 2 parity bits can be accounted for in the fault vector. The weight read out helper function is modified to additionally read the two parity bits in addition to the weight bits and check if the majority of the 2 parity bits and 1 MSB is 1 or 0, and substitute the MSB with it.

5.3.2 Storing the Parity Bits Together with Weight Bits

With the addition of the 2 parity bits with each weight, the question of where to place the 2 parity bits arises. As a starting point, the 2 parity bits are placed next to the MSB of each weight and re-trained. However, this does not provide many meaningful accuracy improvement as can be seen in Figure 5.13. This is to be expected as the weights now consist of 8 bits but the cluster of faults still sits in the center of the SRAM array and thus still has the same effect.

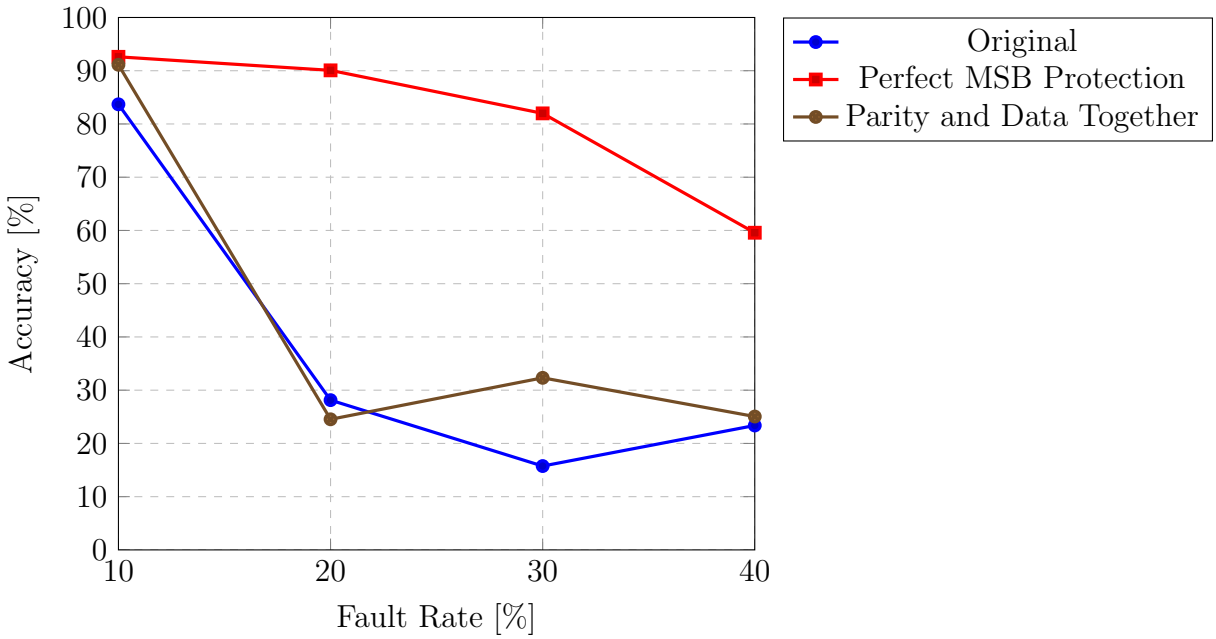
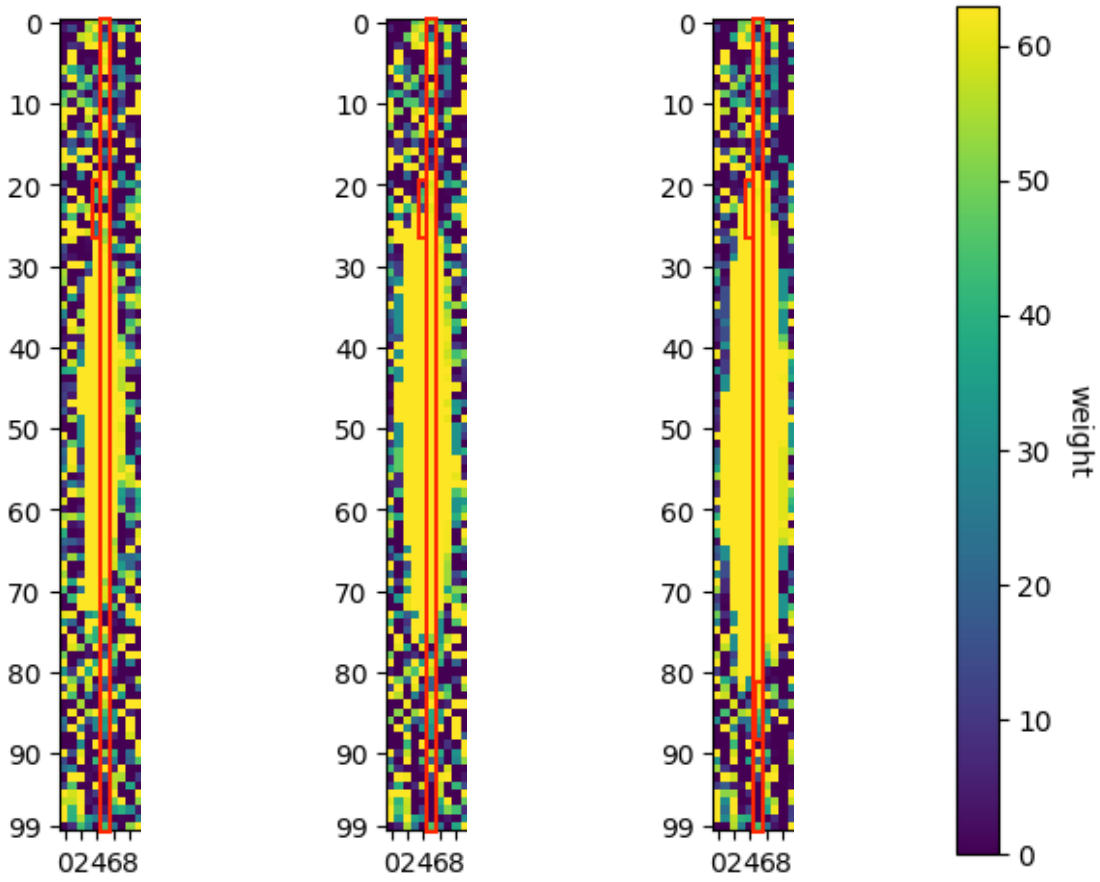


Figure 5.13: Accuracy comparison between the original and perfect MSB protection middle stuck-at 1 fault cluster case, and the 2 protection parity bits stored together with the weight data version of the case.

This case also features a small increase in accuracy at the 30% fault rate. This is another case of the middle cluster expanding to cover weights more evenly and misidentifying digits somewhat less in the 30% case compared to the 20% case, as illustrated in Figure 5.14. From Figure 5.14a, the column 5 associated with the digit 5 in the hidden to output layer is incorrectly mis-identified the most, with the digit 4 being the second most mis-identified. This is due to the fact that the cluster is more even between the columns 5 and 4, and shrinks at the other columns. The accuracy of the network is approximately 20% which make sense as this would be the network recognizing the digits 5 and 4, two out of the ten total digits. At the 30% fault rate in Figure 5.14b, the cluster expands more and covers the other digit columns more evenly. The column 5 and part of column 4 are marked in red to emphasize the cluster affecting adjacent columns 3 and 6 more comparably to 5 and 4, and end up being identified correctly half the each. Finally in Figure 5.14c, at the 40% fault rate the cluster expands and covers columns 5 and 4 the most, where 5 is still mis-identified the most as it's weights at the upper and lower portion of the layer are higher.



(a) Hidden to output layer in the 20% middle cluster fault case.

(b) Hidden to output layer in the 30% middle cluster fault case.

(c) Hidden to output layer in the 40% middle cluster fault case.

(d) Weight value mapping bar.

Figure 5.14: The hidden to output layer with the middle cluster fault pattern under MSB protection scheme.

5.3.3 Storing the Parity Bits Separate from the Weight Bits

Clearly, storing the parity bits close to the weights did not result in accuracies close to the projected perfect MSB protection baseline. In order to remedy the issue, The parity bits need to be placed separately away from their weights so the effect of the stuck-at fault cluster is mitigated. The parity bits can instead be stored along one side of the SRAM arrays. The weights will be stored first on the left side, the 2 parity bits of each weight's

MSB will be stored on the right side, and the middle cluster of stuck-at 1 faults is at the center of this SRAM array.

As seen in Figure 5.15, the accuracy of this configuration is closer to the projected MSB protection baseline as the weights of the middle cluster of faults case have their MSBs protected and recovered by the parity bits sitting on the right side of the SRAM arrays. The additional parity bits also mean that the center of the SRAM array no longer aligns with the center of the weights, thus the effect of the middle cluster is mitigated. This is similar to the distributed weight bit case where the accuracy of the middle fault cluster case was improved due to the MSBs being closer to the edge of the arrays, out of reach of the middle cluster of faults.

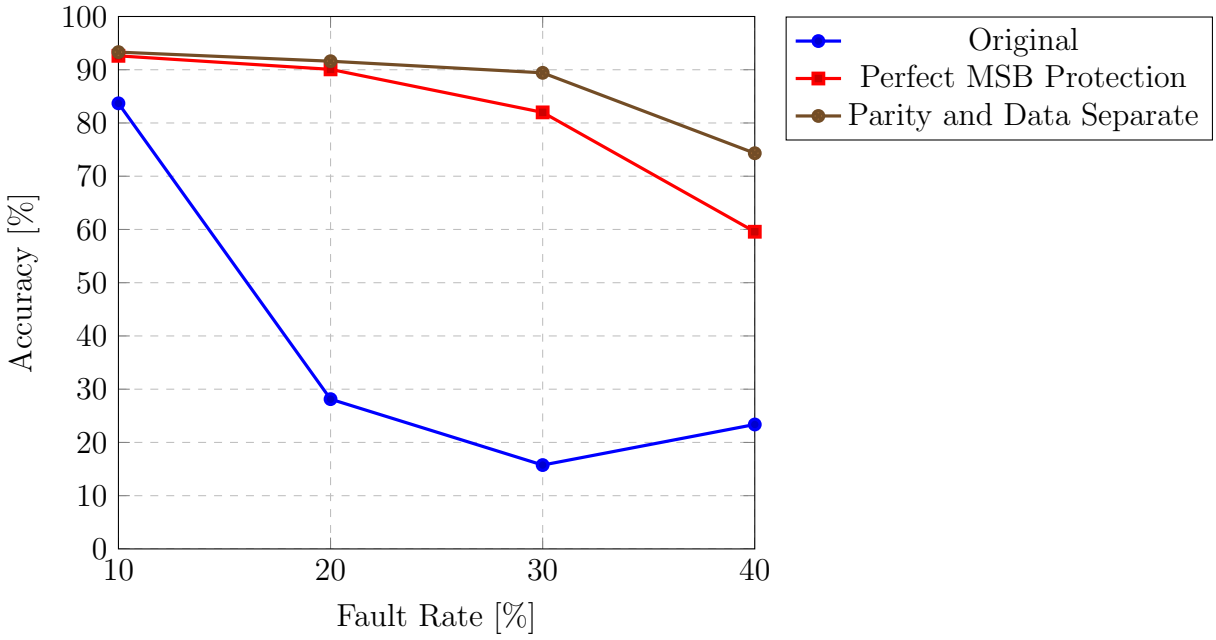


Figure 5.15: Accuracy comparison between the original and perfect MSB protection middle stuck-at 1 fault cluster case, and the separately stored 2 parity bit protection version of the case.

5.3.4 Random Fault Cases

With the effect of the clustered faults against the MSB protection scheme with separated parity bit placement understood, the effect of random stuck-at 1 faults are analyzed as

well. As can be seen in Figure 5.16 , there is a marginal improvement in the 10% to 30% cases, but otherwise the accuracy is approximately the same.

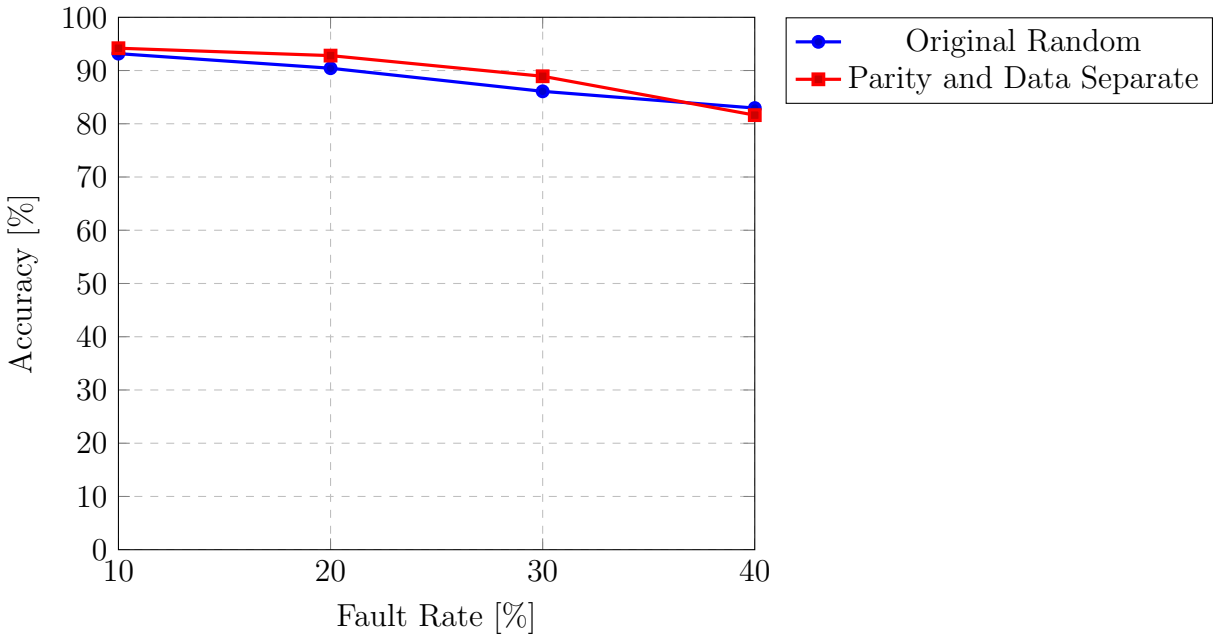


Figure 5.16: Accuracy comparison between the original random stuck-at 1 fault cluster case, and the separately stored 2 parity bit protection version of the case.

5.3.5 Varied Bit Precisions

With the baseline MSB protection accuracies established for the middle fault cluster and the random fault cases, the same experiments are run with different precisions. This will be the 8 bit weights case, and the 4 bit quantized 8 bit and 6 bit cases. The quantization of the 8 and 6 bit weights to 16 values, or 4 bits, maps the weights to a 4 bit quantized value by dividing the range of possible weight values into 16 bins. These quantization bins are assigned a 4 bit value that will map to the value they represent. The 4 bit value is assigned such that they correspond to the original weight. In essence, the maximum value of a 6 bit weight of 63, would be mapped to 15 or 1111, and the lowest value of 0 would be mapped to 0000. The values they represent will be the upper bound of the bin. For example, for the 6 bit case where the maximum value 6 bits can represent is 63, the bin range from 63 to 63 minus one sixteenth of the maximum value will get binned as 63 with

a quantization symbol of 15 or 1111. These quantized values for the 6 bit weight values are summarized in Table 5.3. The width of each quantization bin is equal to each other, and is the total number of values divided by the total number of quantization values. In this case there are 64 possible values divided by 16 quantized values, so each bin spans 4 values. When converting the quantized values to a 6 bit value, the upper bound of their quantization bin are used. The lowest bin, 0000, will be mapped to 0 so that the network can still represent the complete range from 63 down to 0.

Table 5.3: The 6 bit weight bins and their quantized encoding.

Value Bin	63-60	59-56	55-52	51-48	47-44	43-40	39-36	35-32
Quantized	1111	1110	1101	1100	1011	1010	1001	1000
Value Bin	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
Quantized	0111	0110	0101	0100	0011	0010	0001	0000

Additionally, the base accuracies for the quantized and base case as well as the accuracy after re-training a second time is shown in Table 5.4. This is shown since some cases at the low fault rate, 10%, show an improvement slightly above the base accuracy of 93.77%.

Table 5.4: The base accuracies and accuracy with an additional re-training round of the 6 bit case and the 6 to 4 bit quantized cases.

	Base 6 Bit	6 to 4 Bit Quantized
Base Accuracy	93.77%	93.56%
Re-trained Accuracy	94.36%	94.25%

To re-iterate, our cases are as follows for both the middle stuck-at 1 fault cluster case and the random fault case.

- The 6 bit and 8 bit cases
- The 6 bit and 8 bit cases where the weights are quantized down to 16 possible values, or 4 bits.

Starting with the middle cluster case, the 8 bit case performs better than the 6 bit case as seen in Figure 5.17. This is due to the fact that the 8 bit cases widens the hidden to output SRAM array, shifting it's aspect ratio closer to 1:1 such that the MSBs are protected since the middle cluster of faults doesn't affect enough MSBs and parity bits. The 4 bit quantized 8 and 6 bit cases perform marginally better than the baseline perfect MSB protection case.

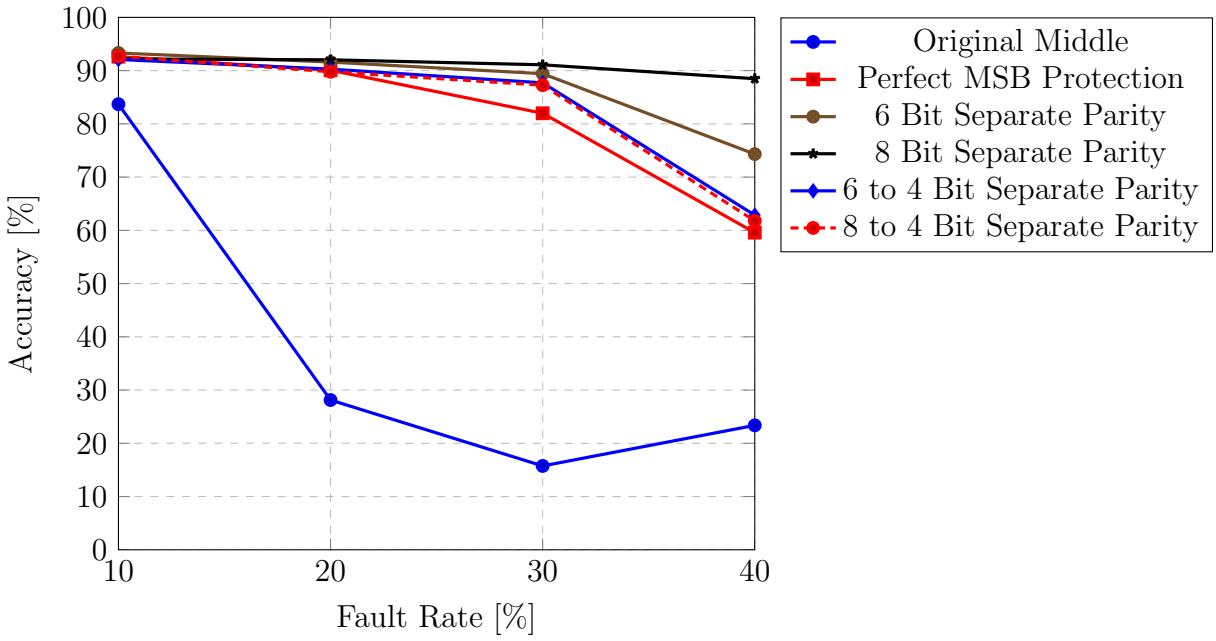


Figure 5.17: Accuracy comparison between the original and perfect MSB protection middle stuck-at 1 fault cluster case, and the separately stored 2 parity bit protection version of the cases with their weights quantized to 4 bits.

As seen in Figure 5.18, in random fault cases the 8 bit case shows a marginal improvement from the 6 bit case at the 20% to 30% fault rates, but are otherwise fairly similar. Examining the 4 bit quantized variants of the 6 and 8 bit cases, the 4 bit quantized variants suffers a larger accuracy drop at the 40% fault rate than the 6 bit case. This is caused by the quantization error from binning the quantized values to the upper bound of the quantization bins. Using the upper bound of the quantization bins results in the stuck-at 1 faults increasing the weights by a larger value than what it would in the standard 6 and 8 bit cases.

In order to remedy this issue, as opposed to using the upper bounds of the quantization

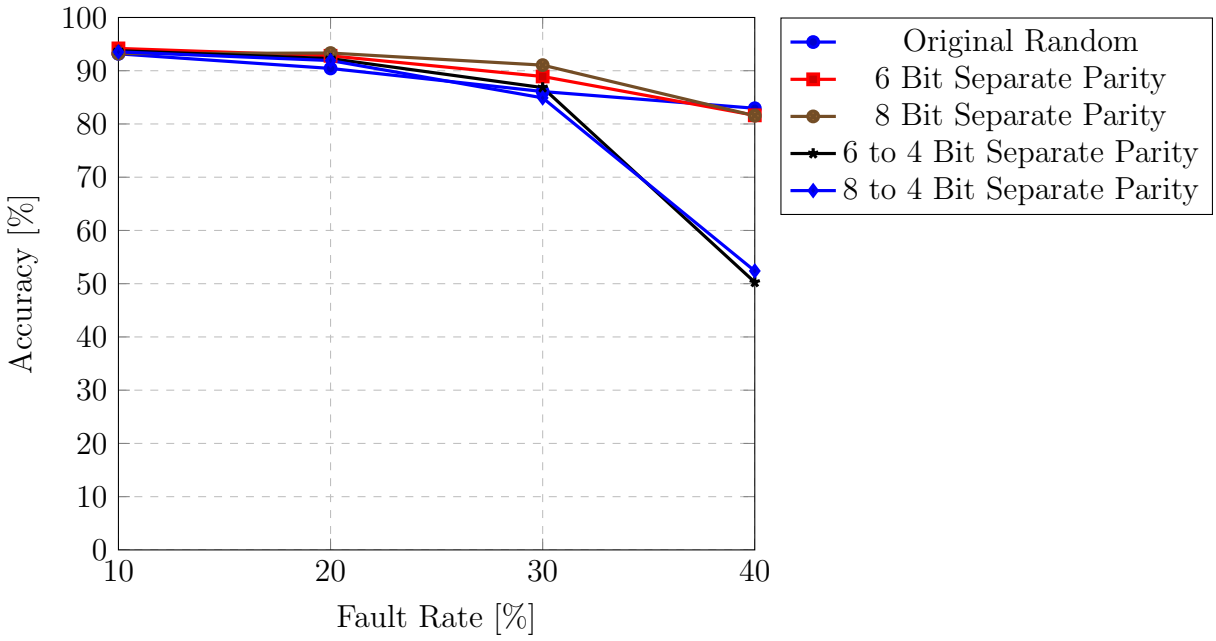


Figure 5.18: Accuracy comparison between the original random stuck-at 1 fault cluster case, and the separately stored 2 parity bit protection version of the case with their weights quantized to 4 bits. The quantization uses the upper bounds of the 16 value quantization bins.

bins the lower bounds of the bins can be used instead for the 16 value or 4 bit quantized values. The resulting re-training accuracies of the quantized 4 bit value 6 and 8 bit cases using the lower bound of the quantization bins are brought closer to the expected values. Note that the top bin is left at the upper bound, or the maximum weight value, so the network can still represent the maximum weight value where it needs to. The resulting accuracy improvement can be seen in Figure 5.19, where the accuracies of the 6 and 8 bit cases as well as the 6 and 8 bit to 4 bit quantized value cases all are marginally above or similar to the nominal base random fault accuracy. Although the 6 and 8 bit to 4 bit quantized value cases perform marginally worse at the 40% fault rate.

5.4 3 by 3 Input to Hidden Layer Average Pooling

As the random stuck-at 1 faults increase the value of the network weights, the network can be re-trained to make use of the faults at the cost of some accuracy. If there were a way

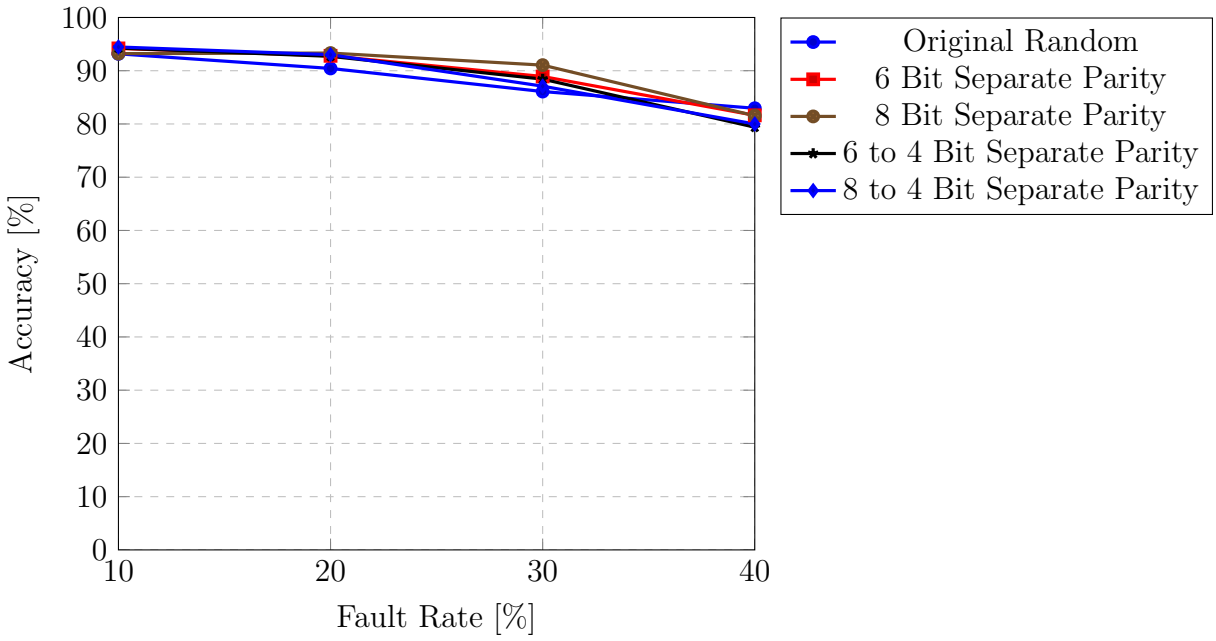


Figure 5.19: Accuracy comparison between the original random stuck-at 1 fault cluster case, and the separately stored 2 parity bit protection version of the case with their weights quantized to 4 bits. The quantization uses the lower bounds of the 16 value quantization bins.

to attenuate the effect of the stuck-at 1 faults, then the weights could be pushed closer to more optimal values for the network's accuracy. Thus to improve the random fault case accuracy, a fault recovery scheme inspired by convolutional neural network's average pooling layer is used.

By attempting to take an average of the weights surrounding a weight index and assuming these adjacent weights were also set to values closer to desired value despite the stuck-at faults would be able to push the resulting used weight closer to the desired value. To achieve this, a 3 by 3 average pooling layer is used on the weights before they are passed on to the sigmoid activation function. This 3 by 3 average pooling layer simply sums all adjacent 3 by 3 weights and divides the sum by 9 in order to average them. The 3 by 3 average pooling layer is applied to the input to hidden layer, as the input is a 2D digit image with spatial locality that could benefit from averaging adjacent weights. Figure 5.20 illustrates the 3 by 3 values that are used to calculate an average to form an output value for use in place of the original. The boxes are the weights which are stored in the SRAM

array, shown as the top left corner of the 22 by 22 image form. The solid boxes would represent the weights corresponding to the 20 by 20 input digit image. The dotted padded boxes around the solid boxes which are not a part of the 20 by 20 digit pixels are added to the SRAM array. This is so that the added edge and corner values can be used in the 3 by 3 averaging operation for weights along the edges and corners. The averaging process is performed to generate a weight for each of the 20 by 20 input digit image pixels as they are sent to the array for multiplication with the weights.

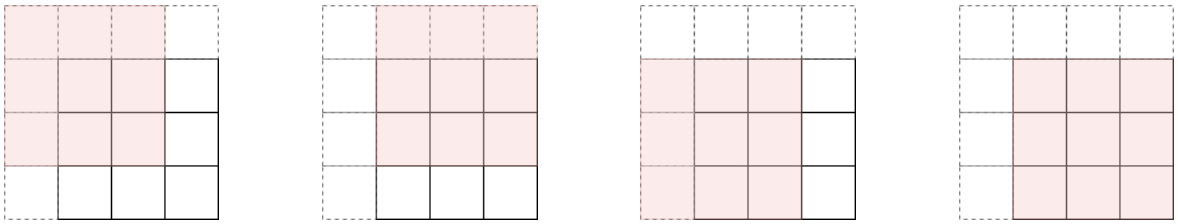


Figure 5.20: Example of averaging 3 by 3 adjacent weights. The light red indicates the 3 by 3 weights used for averaging. Dotted boxes are the added padding weights. Solid boxes are the weights originally from the hidden unit before adding padding weights.

5.4.1 NeuroSim Changes

In order to model the 3 by 3 averaging recovery scheme, all instances of reading values from the input to hidden layer are replaced by a 3 by 3 averaging procedure that sums up the adjacent weights as well as the center weight before dividing by 9 to average them. To account for the edges of the image, the size of the input to hidden layer is expanded or padded to store additional weights along the edge of the layer. This expands the input to hidden layer from a 20 by 20 digit images of 400 weights to a 22 by 22 digit images of 484 weights. Logically, the network still takes input of size 400 as the digit images are still 20 by 20.

In the weight update backpropagation step, the weight update value intended for each of the 400 logical elements of each hidden unit is divided by 9 and applied evenly to the weights of the 3 by 3 pooled group. The confinement of the weight to the -1.0 to 1.0 range step is deferred to after all weights have been updated. This is overcome scenarios such as if a weight were to be set to a value such as -1.0 before hand, the updates before the final update were negative updates that would still leave the weight as -1.0, and the final update was a positive update, the weight would not get fixed to the -1.0 lower limit which results in increasing the weight's value by the final positive update.

5.4.2 Clustered Fault Cases

In the clustered fault cases, the 3 by 3 averaging of adjacent weights is unable to mitigate the concentrated effect of the clustered fault cases. This is shown in Figure 5.21, where the 3 by 3 averaging mechanism is not able to tolerate the top left clustered fault case.

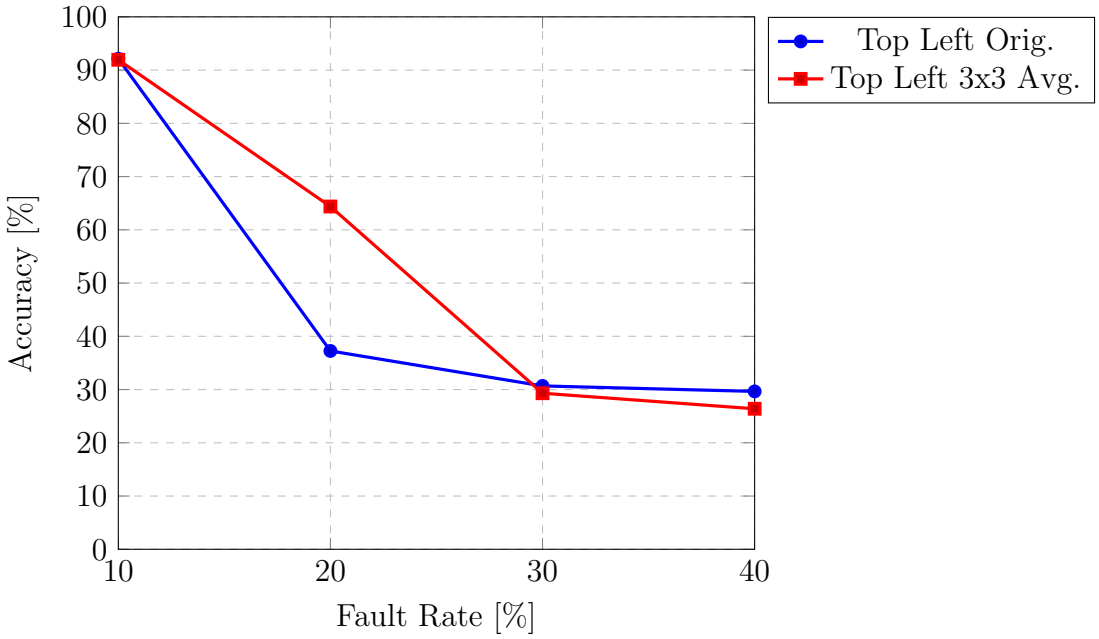


Figure 5.21: Accuracy comparison between the original top left stuck-at 1 fault case, with and without the 3 by 3 average pooling mechanism.

5.4.3 Random Fault Cases

In the random fault cases, the effect of averaging the weights improves the accuracy in the higher fault percentage cases. This shows that the effect of averaging nearby weights can be used successfully to improve the accuracy. Figure 5.22 shows the accuracy improvement of the 5, 6, and 8 bit variants of the 3 by 3 average pooling fault recovery technique. This results in an accuracy improvement of as great as 5% at the 30% and 40% fault rates with the random fault pattern.

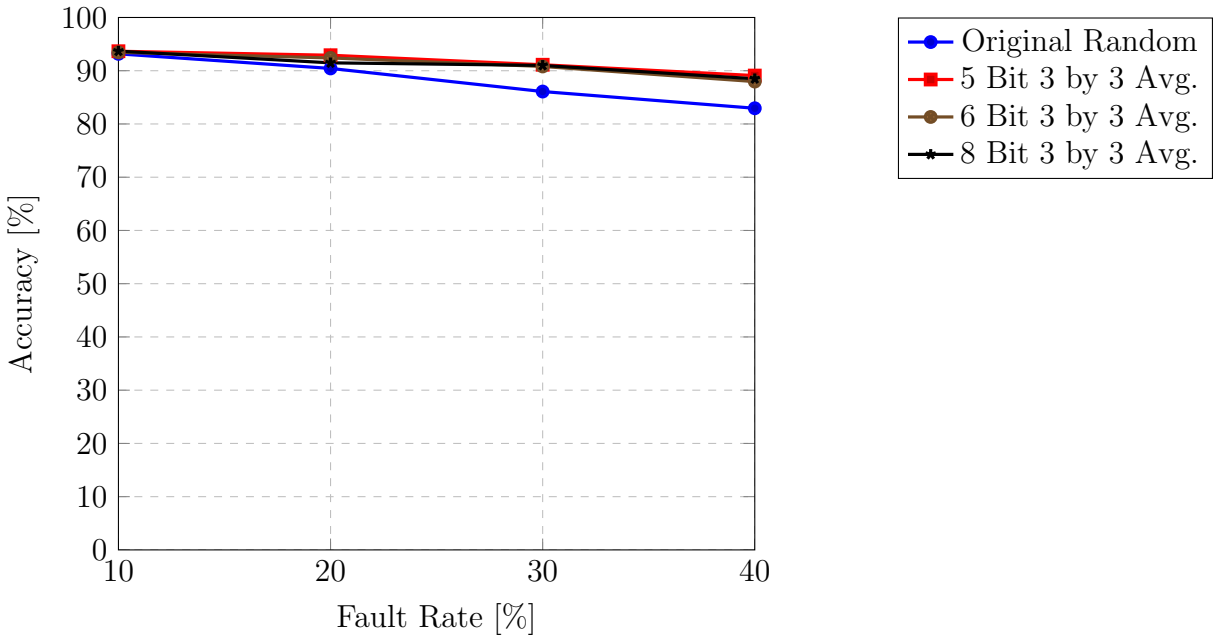


Figure 5.22: Accuracy comparison between the original random stuck-at 1 fault case, and the varied bit precision versions of the 3 by 3 average pooling cases.

5.5 Fault Recovery Scheme Recommendations

The fault recovery schemes described perform well and poorly against different faults and have differing hardware overheads as well. A brief summary of the techniques and their overheads are described before making any final recommendations.

The inverted SRAM array access fault recovery technique performs well against the fault intolerant corner fault cluster cases. However, it is not suited for cases where the faults symmetrically affect the SRAM arrays, such as the middle fault cluster cases. This technique has a negligible storage overhead, and a low hardware overhead.

The distributed weight bit storage fault recovery technique performs well against the middle fault cluster pattern case, and performs the same at the random fault pattern's baseline. However, it doesn't perform as well in the top left fault cluster case where the faults land on the MSBs. This fault recovery technique does not have any additional storage overhead. At most, additional wire routing would be required to route the distributed bits read out from the SRAM array to the adders for weight summation.

The MSB protection fault recovery technique works well against the middle fault cluster

pattern case, and performs slight better in the random fault pattern case at fault rates of 10% to 30%. However since the parity bits are stored along the right side of the SRAM arrays, this fault recovery technique shares the same intrinsic weakness as the distributed weight bit storage technique. If a cluster of faults lands on the two parity bits where the faults are both stuck-at 1 or stuck-at 0 the MSB will no longer be protected. The MSB protection technique has a storage overhead of 25% and marginally requires some hardware to check the MSB and parity bit of each column as the rows are read out.

The 3 by 3 weight average pooling technique improves the random fault pattern case, but does not offer any significant improvement for the clustered cases. This technique uses some additional weights padded around the original 20 by 20, or 400, image size to 22 by 22, or 484, for the 3 by 3 averaging window to average weights stored along the edges. This results in 84 extra weights per column in the input to hidden layer. The 84 extra weights per 100 hidden units over the total number of weights, which amounts to 484 weights per 100 hidden units and 100 hidden by 10 output units results in an overhead of 20.5%.

Table 5.5: A summary of the storage overhead, whether or not additional hardware is required, and fault pattern cases the fault recovery schemes can recover from.

	Overhead	Additional Hardware	Cluster Tolerant	Random Tolerant
Inverted Array Access	Near 0% Constant 4 Bits	Yes	High, 4 of 6. Corners	High
Distributed Weight Bits	Constant 0%	No	High, 3 of 6. MSB Avoided	High
MSB Protection	25%, Increasing with Lower Bit Precision, Decreasing with Higher Bit Precision	Yes	High, 3 of 6. Parity Bit Avoided	High
3 by 3 Average Pooling	Constant 20.5%	Yes	High, 2 of 6. Original Cases	High

In terms of recovered accuracy across the various fault patterns, the distributed weight bit, inverted array access, and MSB protection technique are able to handle a wider variety of fault patterns than the 3 by 3 average weight pooling technique which only improves the

random fault pattern case, as summarized in Table 5.5. Between the distributed weight bit and MSB protection technique, the distributed weight bit technique performs better in the middle cluster fault pattern with 6 bits of precision. The MSB protection technique requires increasing the precision of the weights to 8 bits in order to improve the aspect ratio of the hidden to output SRAM array to keep the cluster of faults away from the parity bits. Between the inverted array access and distributed weight bit storage techniques, the distributed weight bit storage technique isn't able to tolerate clustered fault patterns which cover the MSBs, and the inverted array access technique isn't able to tolerate clustered fault patterns which cover the array symmetrically. However, the distributed weight bit technique does not require addition hardware. Thus, the distributed weight bit technique is recommended overall as it recovers from more fault patterns without an additional storage overhead. In case improving the accuracy of the middle fault cluster pattern is required and the overhead can be tolerated, then 8 bit weight MSB protection technique is recommended. However, if the particular fault pattern the network will encounter is known beforehand then the fault recovery technique that offers the best improvement against the fault pattern should be selected. Particularly, the 3 by 3 average weight pooling technique works best against random faults, and is recommended. It should also be noted that this requires 9 SRAM array reads per weight access, and should only be chosen if the design specifications allow for the higher area, power, and delay required.

An additional alternative is to use some of these techniques together. Without modifying the techniques as they are, the distributed weight bit and 3 by 3 average pooling techniques could be combined to improve fault pattern coverage. The random fault tolerance would remain the same as the distributed bit technique does not vary much under the random fault pattern, and this would provide some partial coverage against the clustered fault patterns. Another combination worth considering with a modification is using the distributed weight bit technique with an inverted bit read technique to improve the cluster fault pattern tolerance. As the distributed weight bit fault pattern is susceptible to corner clusters covering it's MSBs, the accesses to it's bits could be inverted or reversed so the MSBs would be on the opposite side of the SRAM array. This would still be vulnerable to the 3 by 3 cluster fault pattern, and thus improves it's cluster tolerance to 5 of the 6 cases. Furthermore if the specification allows for the increased overhead, these two combinations could be combined with each other to cover 5 of 6 cluster cases, and tolerate random faults. As a final recommendation, as the combination of the distributed weight bit and modified inverted bit read techniques have the lowest overhead and best coverage, it is still recommended in general. If the emerging technology is expected to be subjected to the random fault case at high fault rates and either the accuracy is needed or the application can tolerate the overhead, then it is still recommended to use the 3 by 3 weight averaging

technique. If different fault patterns can be expected and the accuracy of the random case at high fault rates is important, then the combination of the distributed weight bit technique, modified inverted bit read, and 3 by 3 averaging is recommended.

Chapter 6

Conclusion

6.1 Summary

CNFETs are an immature technology with a variety of variation issues. One issue in particular is the presence of stuck-at faults due to metallic carbon nanotubes. This work particularly analyses the effects of stuck-at faults in SRAM arrays in the NeuroSim system. With the effects of the stuck-at faults understood, fault recovery techniques are proposed to mitigate the effect of these stuck-at faults.

Chapter 4 presents a method of modeling faults in NeuroSim before continuing on to analyze the effect of the stuck-at faults in the SRAM arrays of NeuroSim's system. These stuck-at faults are arranged in different fault patterns, and are analyzed with stuck-at 1 faults and then stuck-at 0 faults in order to understand the effect of each. The best and worst fault pattern cases are determined for each stuck-at fault type and a duality between the best and worst cases of the stuck-at 1 and stuck-at 0 faults is found. Additionally, the effect of the number of hidden units is also measured to find that larger networks with more hidden units perform better.

Chapter 5 proposes fault recovery techniques to overcome the effect of the fault patterns on the network. The changes to NeuroSim in order to implement the techniques in the simulation are described. The performance of each technique against the fault patterns is discussed, where each technique has fault patterns they are able to recovery from and those they cannot. In the worst case, the middle stuck-at 1 fault cluster case, the re-training accuracy drops to 23.37% at the 40% fault rate without any fault recovery technique. With a fault recovery technique, the accuracy is improved to 88.08% at the 40% fault rate.

Recommendations are then made based on the fault pattern and overhead tolerance. It is found that distributing the bits of the weights in the SRAM arrays was able to recover from most of the fault pattern cases and effectively has no overhead. Overall, this shows that there is the potential to use emerging technologies such as CNFETs even at the immature early stages of their development.

6.2 Future Work

6.2.1 Stuck-at 1 and Stuck-at 0 Faults

The analysis done in this work analyses the effects of solely stuck-at 1 faults and stuck-at 0 faults. Future work could analyze the effects of both stuck-at 1 and stuck-at 0 faults. To do so, the work would need to consider a variety of stuck-at 1 to stuck-at 0 ratios, as well as distributions within the given fault patterns, giving rise to numerous combinations. To this effect, it would be beneficial to determine a small set of meaningful combinations and stuck-at 1 to stuck-at 0 ratios to analyze with the fault patterns.

6.2.2 Effect of Stuck-at Faults in Other Units

This work analyzes the effect of stuck-at faults in the SRAM arrays of the NeuroSim system. The effect of stuck-at faults being applied to other hardware units could also be analyzed. The hardware units under analysis could be adders, registers, or peripheral hardware such as row decoders to the SRAM array.

6.2.3 Other Accelerator Systems

This work analyzes the effect of stuck-at faults in the NeuroSim which models a MLP neural network. The effect of stuck-at faults could be explored in other accelerators which may target similar or other neural networks. Some examples of accelerators which can target other neural networks such as DNNs or CNNs are NVIDIA Deep Learning Accelerator (NVDLA), Eyeriss, and the Tensor Processing Unit (TPU). Future work could model stuck-at faults in these accelerators against different modern DNNs and CNNs applications such as AlexNet, SqueezeNet, or ResNet-50. The stuck-at faults could also be analyzed in different hardware units.

References

[1] Amin Ansari, Shantanu Gupta, Shuguang Feng, and Scott Mahlke. Zerehcache: Armoring cache architectures in high defect density technologies. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 100–110, 2009.

[2] Debjit Chattopadhyay, Izabela Galeska, and Fotios Papadimitrakopoulos. A route for bulk separation of semiconducting from metallic single-wall carbon nanotubes. *Journal of the American Chemical Society*, 125(11):3370–3375, 2003.

[3] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 6–1. IEEE, 2017.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[5] Chen, Pai-Yu and Peng, Xiaochen and Luo, Yandong and Yu Shimeng. Neurosimv3.0_user_mannual.pdf. https://github.com/neurosim/MLP_NeuroSim_V3.0/blob/master/documents/NeuroSimV3.0_user_mannual.pdf.

[6] J-P Colinge. Multigate transistors: Pushing moore's law to the limit. In *2014 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, pages 313–316. IEEE, 2014.

[7] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[8] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[9] Seyyed Ashkan Ebrahimi, Mohammad Reza Reshadinezhad, and Ali Bohlooli. A new design method for imperfection-immune cnfet-based circuit design. *Microelectronics Journal*, 85:62–71, 2019.

[10] Shu-Jen Han, Jianshi Tang, Bharat Kumar, Abram Falk, Damon Farmer, George Tulevski, Keith Jenkins, Ali Afzali, Satoshi Oida, John Ott, et al. High-speed logic integrated circuits with solution-processed self-assembled carbon nanotubes. *Nature nanotechnology*, 12(9):861–865, 2017.

[11] Gage Hills, Daniel Bankman, Bert Moons, Lita Yang, Jake Hillard, Alex Kahng, Rebecca Park, Marian Verhelst, Boris Murmann, Max M Shulaker, et al. Trig: Hardware accelerator for inference-based applications and experimental demonstration using carbon nanotube fets. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–10, 2018.

[12] Gage Hills, Christian Lau, Andrew Wright, Samuel Fuller, Mindy D Bishop, Tathagata Srimani, Pritpal Kanhaiya, Rebecca Ho, Aya Amer, Yosi Stein, et al. Modern microprocessor built from complementary carbon nanotube transistors. *Nature*, 572(7771):595–602, 2019.

[13] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[14] Engin Ipek, Jeremy Condit, E Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building resilient systems from unreliable nanoscale memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, volume 10, 2010.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[16] Pritpal S Kanhaiya, Gage Hills, Dimitri A Antoniadis, and Max M Shulaker. Disc-fets: Dual independent stacked channel field-effect transistors. *IEEE Electron Device Letters*, 39(8):1250–1253, 2018.

[17] Robert W Keyes. Physical limits of silicon transistors and circuits. *Reports on Progress in Physics*, 68(12):2701, 2005.

[18] Bharat Kumar, Abram L Falk, Ali Afzali, George S Tulevski, Satoshi Oida, Shu-Jen Han, and James B Hannon. Spatially selective, high-density placement of polyfluorene-sorted semiconducting carbon nanotubes in organic solvents. *ACS nano*, 11(8):7697–7701, 2017.

[19] Tianjian Li, Li Jiang, Xiaoyao Liang, Qiang Xu, and Krishnendu Chakrabarty. Defect tolerance for cnfet-based srams. In *2016 IEEE International Test Conference (ITC)*, pages 1–9. IEEE, 2016.

[20] Tianjian Li, Feng Xie, Xiaoyao Liang, Qiang Xu, Krishnendu Chakrabarty, Naifeng Jing, and Li Jiang. A novel test method for metallic cnts in cnfet-based srams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(7):1192–1205, 2015.

[21] Xuefei Li, Zhuoqing Yu, Xiong Xiong, Tiaoyang Li, Tingting Gao, Runsheng Wang, Ru Huang, and Yanqing Wu. High-speed black phosphorus field-effect transistors approaching ballistic limit. *Science advances*, 5(6):eaau3194, 2019.

[22] Chenchen Liu, Miao Hu, John Paul Strachan, and Hai Li. Rescuing memristor-based neuromorphic design with high defects. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[23] Lijun Liu, Jie Han, Lin Xu, Jianshuo Zhou, Chenyi Zhao, Sujuan Ding, Huiwen Shi, Mengmeng Xiao, Li Ding, Ze Ma, et al. Aligned, high-density semiconducting carbon nanotube arrays for high-performance electronics. *Science*, 368(6493):850–856, 2020.

[24] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[25] Paul L McEuen, Michael S Fuhrer, and Hongkun Park. Single-walled carbon nanotube electronics. *IEEE transactions on nanotechnology*, 1(1):78–85, 2002.

[26] Steven G Noyce, James L Doherty, Zhihui Cheng, Hui Han, Shane Bowen, and Aaron D Franklin. Electronic stability of carbon nanotube transistors under long-term bias stress. *Nano letters*, 19(3):1460–1466, 2019.

[27] Rebecca S Park, Gage Hills, Joon Sohn, Subhasish Mitra, Max M Shulaker, and H-S Philip Wong. Hysteresis-free carbon nanotube field-effect transistors. *ACS nano*, 11(5):4785–4791, 2017.

[28] Nishant Patil, Albert Lin, Jie Zhang, Hai Wei, Kyle Anderson, H-S Philip Wong, and Subhasish Mitra. Vmr: Vlsi-compatible metallic carbon nanotube removal for imperfection-immune cascaded multi-stage digital logic circuits using carbon nanotube fets. In *2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–4. IEEE, 2009.

[29] Chenguang Qiu, Zhiyong Zhang, Mengmeng Xiao, Yingjun Yang, Donglai Zhong, and Lian-Mao Peng. Scaling carbon nanotube complementary transistors to 5-nm gate lengths. *Science*, 355(6322):271–276, 2017.

[30] Ali Razavieh, Peter Zeitzoff, and Edward J Nowak. Challenges and limitations of cmos scaling for finfet and beyond architectures. *IEEE Transactions on Nanotechnology*, 18:999–1004, 2019.

[31] Stuart Schechter, Gabriel H Loh, Karin Strauss, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. *ACM SIGARCH Computer Architecture News*, 38(3):141–152, 2010.

[32] Catherine D Schuman, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

[33] Frank Schwierz. Graphene transistors. *Nature nanotechnology*, 5(7):487–496, 2010.

[34] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A Rivers, and Hsien-Hsin S Lee. Safer: Stuck-at-fault error recovery for memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 115–124. IEEE, 2010.

[35] Jia Si, Donglai Zhong, Haitao Xu, Mengmeng Xiao, Chenxi Yu, Zhiyong Zhang, and Lian-Mao Peng. Scalable preparation of high-density semiconducting carbon nanotube arrays for high-performance field-effect transistors. *ACS nano*, 12(1):627–634, 2018.

[36] T Srimani, G Hills, X Zhao, D Antoniadis, JA Del Alamo, and MM Shulaker. Asymmetric gating for reducing leakage current in carbon nanotube field-effect transistors. *Applied Physics Letters*, 115(6):063107, 2019.

[37] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[38] Stuart Thomas. Nanosheet fets at 3 nm. *Nature Electronics*, 1(12):613–613, 2018.

[39] Zhen Wang, Mark Karpovsky, and Ajay Joshi. Influence of metallic tubes on the reliability of cntfet srams: error mechanisms and countermeasures. In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, pages 359–362, 2011.

[40] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. *ACM SIGARCH computer architecture news*, 36(3):203–214, 2008.

[41] Tony F Wu, Haitong Li, Ping-Chen Huang, Abbas Rahimi, Gage Hills, Bryce Hodson, William Hwang, Jan M Rabaey, H-S Philip Wong, Max M Shulaker, et al. Hyperdimensional computing exploiting carbon nanotube fets, resistive ram, and their monolithic 3d integration. *IEEE Journal of Solid-State Circuits*, 53(11):3183–3196, 2018.

[42] Peide Ye, Thomas Ernst, and Mukesh V Khare. The last silicon transistor: nanosheet devices could be the final evolutionary step for moore's law. *IEEE Spectrum*, 56(8):30–35, 2019.