ABSTRACT

Title of dissertation: SPINTRONICS-BASED ARCHITECTURES FOR NON-VON NEUMANN COMPUTING

Ankit Mondal
Doctor of Philosophy, 2020

Dissertation directed by: Professor Ankur Srivastava
Department of Electrical and
Computer Engineering

The scaling of transistor technology in the last few decades has significantly impacted our lives. It has given birth to different kinds of computational workloads which are becoming increasingly relevant. Some of the most prominent examples are Machine Learning based tasks such as image classification and pattern recognition which use Deep Neural Networks that are highly computation and memory-intensive. The traditional and general-purpose architectures that we use today typically exhibit high energy and latency on such computations. This, and the apparent end of Moore's law of scaling, has got researchers into looking for devices beyond CMOS and for computational paradigms that are non-conventional. In this dissertation, we focus on a spintronic device, the Magnetic Tunnel Junction (MTJ), which has demonstrated potential as cache and embedded memory. We look into how the MTJ can be used beyond memory and deployed in various non-conventional and non-von Neumann architectures for accelerating computations or making them energy-efficient.

First, we investigate into Stochastic Computing (SC) and show how MTJs can

be used to build energy-efficient Neural Network (NN) hardware in this domain. SC is primarily bit-serial computing which requires simple logic gates for arithmetic operations. We explore the use of MTJs as Stochastic Number Generators (SNG) by exploiting their probabilistic switching characteristics and propose an energy-efficient MTJ-SNG. It is deployed as part of an NN hardware implemented in the SC domain. Its characteristics allow for achieving further energy efficiency through NN weight approximation, towards which we develop an optimization problem.

Next, we turn our attention to analog computing and propose a method for training of analog Neural Network hardware. We consider a resistive MTJ crossbar architecture for representing an NN layer since it is capable of in-memory computing and performs matrix-vector multiplications with O(1) time complexity. We propose the on-chip training of the NN crossbar since, first, it can leverage the parallelism in the crossbar to perform weight update, second, it allows to take into account the device variations, and third, it enables avoiding large sneak currents in transistor-less crossbars which can cause undesired weight changes.

Lastly, we propose an MTJ-based non-von Neumann hardware platform for solving combinatorial optimization problems since they are NP-hard. We adopt the Ising model for encoding such problems and solving them with simulated annealing. We let MTJs represent Ising units, design a scalable circuit capable of performing Ising computations and develop a reconfigurable architecture to which any NP-hard problem can be mapped. We also suggest methods to take into account the non-idealities present in the proposed hardware.

SPINTRONICS-BASED ARCHITECTURES FOR NON-VON NEUMANN COMPUTING

by

Ankit Mondal

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Ankur Srivastava, Chair/Advisor
Professor Bruce Jacob
Professor Manoj Franklin
Professor Donald Yeung
Professor Mohammad Hajiaghayi

Dedication

To my parents, for their love and constant support.

Acknowledgments

I would like to thank several people who have been a part of my life, either in my professional environment or otherwise, in the last few years and with me in this incredible journey. First of all, I would thank my PhD advisor Prof. Ankur Srivastava for having always patiently guided me in my work. I owe him my gratitude for teaching me the basics of doing research. His invaluable advice in several circumstances has helped me to navigate through difficult situations. His knack for looking for the right research problems and the best ways to approach them always kept me motivated and hopeful.

I would like to thank my committee members Profs. Donald Yeung, Mohammad Hajiaghayi, Bruce Jacob and Manoj Franklin for their willingness to sit through and evaluate my work, and Profs. Franklin and Jacob for their useful feedback after my candidacy exam.

I would also express my gratitude towards my colleagues and friends in the lab, both in the past and in the present. During the start of my PhD, I could always look to my seniors Chongxi Bao, Zhiyuan Yang, Yang Xie and Yuntao Liu for guidance as they helped bridge any gap in communication and understanding with my advisor. Thanks are also due to Abhishek Chakraborty and Mike Zuzak whose experience I could count on in situations that were new to me. Them, and the new members of our lab Daniel Xing and Nina Jacobsen, and my other friends in College Park made sure that graduate student life wasn't monotonous.

Lastly, I would like to acknowledge the moral support and love that my parents have provided me during my time away from home and their faith in me.

Table of Contents

Dedication ii

Acknowledgement iii

Table of Contents iv

List of Tables vii

List of Figures viii

List of Abbreviations x

1 Introduction 1
- 1.1 Focus and outline of thesis . . . 5

2 Preliminaries 7
- 2.1 Artificial Neural Network Architecture . . . 7
  - 2.1.1 Training of Neural Networks . . . 9
- 2.2 Spintronics and the Magnetic Tunnel Junction . . . 11
  - 2.2.1 Magnetic Tunnel Junction . . . 12
  - 2.2.2 Other spintronic devices . . . 14
  - 2.2.3 Memristive devices . . . 14
- 2.3 Non-conventional and Non-von Neumann Computing . . . 15
  - 2.3.1 The Resistive Crossbar Architecture . . . 18
- 2.4 Non-conventional computing paradigms . . . 20
  - 2.4.1 Approximate Computing . . . 21
  - 2.4.2 Probabilistic Computing . . . 21
  - 2.4.3 Stochastic Computing . . . 21

3 Stochastic Computing with MTJ for Neural Networks 23
- 3.1 Introduction . . . 23
- 3.2 Preliminaries . . . 25
  - 3.2.1 Stochastic Computing . . . 25
  - 3.2.2 Computational units in SC . . . 26
- 3.3 MTJ-based Stochastic Computing . . . 27
  - 3.3.1 Characteristics of Magnetic Tunnel Junctions . . . 28
  - 3.3.2 MTJ as a Stochastic Number Generator . . . 29
  - 3.3.3 Proposed Biased MTJ-SNG . . . 30
  - 3.3.4 Comparison with CMOS-based SNG . . . 31

3.4 Energy Efficient MTJ-based NN Implementation . . . 32
3.4.1 NN implementation in the SC/ISC domain . . . 33
3.4.2 Problem Formulation . . . 34
3.4.3 Optimizing a 1-layer NN . . . 35
3.4.4 Optimizing 2-layer NNs . . . 37
3.5 Regularization and Constraints for Classification problems . . . 38
3.5.1 Regularization . . . 38
3.5.2 Classification Specific Customization . . . 41
3.6 Simulation Methodology and Results . . . 43
3.6.1 Evaluation setup . . . 43
3.6.2 Results . . . 44
3.7 Conclusion . . . 50

4 In-situ Training of MTJ Neural Network Crossbar 51
4.1 Introduction . . . 51
4.2 Background . . . 54
4.2.1 Crossbar Architecture for Neural Networks . . . 54
4.2.2 Related Work . . . 55
4.3 MTJ Crossbar based Neural Networks . . . 56
4.3.1 Training Binary Networks . . . 56
4.3.2 The Motivation for In-situ Training . . . 57
4.3.3 Network Binarization and MTJ as a synapse . . . 58
4.4 In-situ Training of the NN Crossbar . . . 59
4.4.1 Overview of Operations . . . 60
4.4.2 Stochastic Learning of an MTJ Synapse . . . 61
4.4.3 The 1T1R Architecture . . . 64
4.4.3.1 Updating the crossbar . . . 65
4.4.3.2 Control circuits . . . 67
4.4.4 The 1R Architecture . . . 68
4.4.4.1 Two-phase update . . . 69
4.4.4.2 Four-phase Update . . . 70
4.4.5 Multi-Layer NNs . . . 72
4.5 Training of Restricted Boltzmann Machines . . . 74
4.5.1 Basics of RBM . . . 74
4.5.2 Deep Belief Networks . . . 77
4.5.3 Adaptation of the Contrastive Divergence algorithm . . . 78
4.5.4 Training of RBM MTJ crossbar . . . 80
4.5.5 MTJs for hidden units . . . 83
4.6 Simulation Setup and Results . . . 86
4.6.1 Neural Networks . . . 87
4.6.1.1 Methodology . . . 87
4.6.1.2 Results . . . 88
4.6.2 Deep Belief Networks . . . 92
4.7 Discussion . . . 95
4.8 Conclusion . . . 98

5 MTJ-based Ising Model Architecture 100
5.1 Introduction and Related Work . . . 100
5.1.1 Related Work . . . 101
5.1.2 Our contribution . . . 102
5.2 The Ising Model . . . 103
5.3 Ising-FPGA Framework . . . 105
5.3.1 Finding local optimum in the Ising model . . . 105
5.3.2 MTJ as an Ising spin unit . . . 106
5.3.3 MTJ-based Ising-FPGA cell . . . 110
5.3.4 Splitting inputs to multiple cells . . . 112
5.4 Architecture of the Ising-FPGA . . . 114
5.4.1 Architecture of an FPGA . . . 115
5.4.2 Reconfigurable Ising model hardware . . . 116
5.4.3 Signal Degradation and Recovery . . . 118
5.5 Ising graphs of NP-hard problems . . . 121
5.5.1 Maximum Cut . . . 121
5.5.2 Travelling Salesman Problem . . . 121
5.6 Simulation Setup and Results . . . 123
5.6.1 Methodology . . . 123
5.6.2 Results . . . 124
5.7 Discussion . . . 127
5.8 Conclusion . . . 128

6 Conclusion and Future Work 129
6.1 Ising Graph simplification . . . 129
6.2 Neuromorphic Computing with Spintronics . . . 130

Bibliography 132

List of Tables

3.1 Comparison of Normal and Biased MTJ-SNG 31
3.2 Notations for problem formulation of 1-layer NN 36
3.3 Variation of 1-layer network energy and classification error rate on the MNIST test dataset 45
3.4 Results for the MNIST 2-layer network for select values of error threshold of the outer layer 46

4.1 The write phase. Signs of x, δ, and ΔW, required change in weight W and conductance G, and the desired direction of switching of MTJ Synapse 60
4.2 Boundary values of the parameters in the weight update eqn. (2.3) and their counterpart in probabilistic switching of MTJ. 63
4.3 The coefficients that fit the model for both $AP \rightarrow P$ and $P \rightarrow AP$ switching 63
4.4 4-phase weight update for the 1R configuration in fig 4.6(c) 71
4.5 The stages of CD training when different currents (fig. 4.10(b)) operate on the hidden units and the switches that are active 84
4.6 Notations for MTJ curve fitting 84
4.7 Classification error rates for the 3 datasets with various NN and crossbar architectures under different training scenarios. 89
4.8 Misclassification rates of NNs with stochastic training of 1T1R and 1R architectures under different levels of device variations 89
4.9 Misclassification rates with hidden layers trained as RBM for the MNIST and WBCD datasets, for different levels of device variations. 93

5.1 Configuration of Ising cells as per their level. 114
5.2 Descriptions of graphs for Maxcut simulations. 125
5.3 Ising-FPGA hardware usage for Max Cut. 125
5.4 Ising-FPGA hardware usage for TSP. 126
5.5 Results of Ising simulations for TSP. 127

List of Figures

1.1 Major computing frameworks driving the tech sector 2

2.1 Schematic of a neuron and the *tanh* activation function 8
2.2 Single-layer and 2-layer NNs 9
2.3 The MTJ and Spin-Torque Transfer switching from P→AP and AP→P 12
2.4 MTJ $AP \rightarrow P$ switching probability as a function of t and I 13
2.5 Structure of an $M \times N$ resistive crossbar 19
2.6 Opportunities to overcome the hurdles presented by the slowing down of Moore's law as 3 paths, not necessarily meant to be used mutually exclusively. 20

3.1 Components used in SC and direction of flow of data 26
3.2 Scaled addition in SC, Integral SC (ISC) representation, and Multiplication in ISC 26
3.3 The BMS and variation of energy with value of SN p with and without BMS 30
3.4 Neuron implementation in ISC and Schematic of 1-layer NN 34
3.5 Regularization function types 1,2 and 3 respectively and their derivatives 40
3.6 Flow chart showing the process and network optimization and characterization 44
3.7 Energy vs classification error rate curve for the MNIST dataset 1-layer NN 46
3.8 Plot of classification error rate against energy for 2-layer NN of MNIST dataset with AV constraint. 47
3.9 Trade-off between network Energy and classification error rate for the Wine Quality test dataset 48
3.10 Energy v/s inaccuracy in classification for the SONAR dataset 49

4.1 A 2×2 crossbar with selection transistors 58
4.2 Synaptic weights and activation function in each column of the crossbar. Comparison of the output characteristics of the inverter and the actual *tanh* function. 60
4.3 Comparison of the linear model and desired switching probabilities . 63
4.4 The 1T1R crossbar. (a) Schematic (b) Read & write phases signals . 65
4.5 Circuit of the crossbar's (a) Input, and (b) Output CLs, and (c) Write phase signals timing diagram 68

4.6 Alternate current paths in the 1R structure with 2-phase write strategy 69
4.7 Circuits for gradient descent and backpropagation 72
4.8 Schematics of RBM and DBN 76
4.9 (a) Crossbar implementation structure of the RBM with MTJs as synapses and hidden units. (b) Signals during the 5 stages of the CD update cycle. 80
4.10 (a) DBN crossbar structure - 4×4 and 4×3 crossbars for the 1^{st} and 2^{nd} hidden layers. (b) Circuit of MTJ as RBM hidden unit connected to the respective Control Logic. 84
4.11 Switching probabilities of the MTJ meant to store RBM hidden unit's states 86
4.12 NN training error with different extents of device variations on the 1R crossbar for 2 datasets. 90
4.13 Comparison of error during training of the 1R crossbar with 2-phase and 4-phase update schemes for 2 datasets. No variations assumed. . 90
4.14 Progress of training with RBMs as hidden layers 94
4.15 Comparison of standard CD and 2-stage CD, and histogram showing distribution of inputs to the sigmoid activation function of the 1^{st} hidden layer 95

5.1 Ising graph and Ising energy landscape 104
5.2 Switching probabilities of the MTJ with $2ns$ pulse width 107
5.3 Different stages of an iteration in the process of finding the ground state of an Ising model 109
5.4 The proposed Ising spin cell 111
5.5 The Modified Ising cell and multi-level Ising cells 113
5.6 The FPGA architecture 115
5.7 The BLIF File Generator and an excerpt from a BLIF file 117
5.8 Routing of nets in the Ising-FPGA 119
5.9 Signal degradation model for paths from a last level cell (source) to level A cells (destinations). 120
5.10 Arrangement of Ising spin units for a 5-city TSP 122
5.11 Steps performed in the simulations. We start with the Graph Generator and end with the Stochastic LLG simulations. 123
5.12 Max cut values (normalized) from the Ising simulations for the 4 graphs, with 2 different values of Ising cell fan-in (I) used for G2. . . 126
5.13 Average (over 20 runs) no. of valid tours found in a run. 127

List of Abbreviations

ANN	Artificial Neural Network
SC	Stochastic Computing
DNN	Deep Neural Network
ML	Machine Learning
AI	Artificial Intelligence
MTJ	Magnetic Tunnel Junction
STT-MRAM	Spin-Transfer Torque Magnetic RAM
ASL	All Spin Logic
MAC	Multiply and Accumulate
ADC	Analog to Digital Converter
RRAM	Resistive RAM
SNG	Stochastic Number Generator
SN	Stochastic Number
DBN	Deep Belief Network
RBM	Restricted Boltzmann Machine
MLP	Multi-layer Perceptron
ISC	Integral Stochastic Computing
FSM	Finite State Machine
P/AP	Parallel/Anti-Parallel
BMS	Biased MTJ-SNG
LFSR	Linear Feedback Shift Register
MSE	Mean Square Error
AV	Absolute Value
CS	Classification Specific
SNN	Spiking Neural Network
STDP	Spike-Timing Dependent Plasticity
CD	Contrastive Divergence
RRAM	Resistive RAM
1T1R	1-Transistor 1-Resistor
CL	Control Logic
LLG	Landau-Lifschitz-Gilbert
DA	Differential Amplifier
SFT	Supervised Fine Tuning
FPGA	Field Programmable Gate Array
VTR	Verilog to Routing
VPR	Versatile Place and Route
CLB	Configurable Logic Block
BLE	Basic Logic Element
LUT	Look-up Table
SB	Switch Box
BLIF	Berkeley Logic Interchange Format
BFG	BLIF File Generator
TSP	Travelling Salesman Problem
CWF	Channel Width Factor

Chapter 1: Introduction

The last few decades have witnessed significant growth in computing capabilities and electronics which has significantly impacted our life - how we live and how we work. This has been possible primarily because of the scaling of technological devices (transistors), which brought about improvements in speed, power consumption and cost. The ability to sustain the operation of billions of transistors in a small area has resulted in the proliferation of consumer electronic goods. That in turn has led to the birth of several computational frameworks which form a major fraction of today's digital workload. Fig. 1.1 mentions some of the frameworks and workloads of today, which are sometimes interconnected or interdependent.

One of the most prominent concepts which is driving the growth of several sectors in the industry is that of Machine Learning (ML) and Artificial Intelligence (AI). The capability of the human brain to learn and solve complex problems have inspired advancements in areas of neuroscience, AI and ML. Decades of research in Artificial Neural Networks (ANNs), despite our limited understanding of biological Neural Networks (NNs), have shown promising results in applications such as pattern recognition, image classification and Natural Language Processing [109]. Deep Neural Networks (DNNs), which are NNs having several layers cascaded, have thus become a popular choice for such ML-based applications. The unprecedented suc-

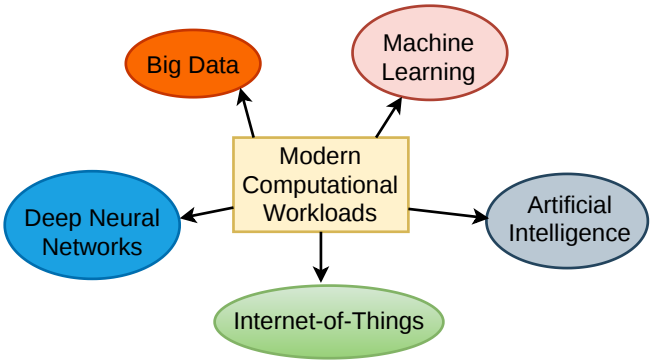


Figure 1.1: Major computing frameworks driving the tech sector

cess in these tasks has however been at the cost of massive computations on von Neumann architectures exhibiting high energy or area requirements, or both. An example would be the IBM Blue Gene supercomputers which have tens of thousands of processors and consume power in the order of a Megawatt [46]. Such resource-hungry characteristics of DNNs makes their implementation prohibitive on platforms with limited capacity such as mobile devices and embedded systems. And this has motivated researchers to think beyond what is traditional in terms of hardware platforms for NNs. The emergence of novel devices and special-purpose architectures encourages a shift from conventional digital hardware for implementing neural algorithms [121].

The saturation of technological scaling and its diminishing returns (in terms of voltage and clock frequency scaling and integration density) is signalling an end to the Moore's law. The search for device technologies alternative to CMOS has been going for quite a long time. And while there is no clear successor yet which can replace it throughout, several candidates have emerged with their own strengths and weaknesses, demonstrating superiority in some domain/application. The more promising among these are spintronic devices and memristors, which offer characteristics such as non-volatility, near-zero leakage and high integration density. Memory

chips based on these technologies are either already commercially available or close to market [107].

The rise of other device technologies is however not sufficient to keep the momentum in the growth of computing. Dedicated processing units are increasingly being deployed to speed up execution. For example, most modern consumer electronic devices, such as smartphones, have special processors (ASICs) that take some load off the CPU(s) for applications such as video-processing. The same goes for server processors which leverage GPUs to perform computations in parallel. A significant and ever-increasing bottleneck in modern computing is actually the gap between memory and logic. The execution speed of the processor cannot exceed the rate at which instructions and data are fetched from memory. Although the aforementioned accelerators improve computational throughput, the fundamental problem of the memory bottleneck still remains.

On the horizon are circuits and higher-level architectures that are more energy efficient than von-Neumann computers by departing from the traditional concept of sequential flow of program execution. These beyond von-Neumann architectures are adapted to specific computing requirements and designed to accelerate increasingly prominent workloads. One common technique called near-memory computing brings memory closer to logic, but the processing units are still distinct from memory arrays [55]. Another form of computing which is truly non-von Neumann is in-memory computing where the processing on data is done at the same place where it is stored, thereby completely eliminating data movement. Emerging resistive memory devices are a good candidate for this framework because their variable conductance can be leveraged to perform multiplications and additions using Ohm's and Kirchhoff's law.

This kind of computing is inherently analog in nature and has led to the development of massively parallel accelerators for a wide range of applications.

An additional effect of the shrinking technology is the increased difficulty in ensuring error-free computing. As per the 2007 report of the International Technology Roadmap for Semiconductors (ITRS), relaxing the stringent requirements of correctness can result in significant savings in manufacturing costs. This, and the ability of deep learning and big-data applications to tolerate minor errors in computations, has increased the relevance of imprecise computing methods. It refers to allowing some deviations in calculations from the specifications by harnessing noise and error to achieve energy-efficiency. The most popular category of such methods is Approximate Computing which aims to save energy spent in computing by reducing the accuracy or probability of correctness of answers. Neuromorphic computing, wherein operations are performed in a manner similar to how the brain and its neurons function, is another non-von Neumann framework which leverages imprecise computing paradigms.

Another class of problems that has always been computationally intractable is combinatorial optimization, which is encountered in several applications in daily life. It involves choosing an optimal configuration of the state variables from a large number of possible ones in a problem with a discrete solution space. It is well known that our traditional von Neumann computers are not well suited to solve such NP-hard problems [27] because a large no. of calculations need to be done for solving such problems. A better way is to map the problem to a model which can be used to find a local optimum via natural computing techniques. Instead of solving step-by step, the system representing the model is left to itself and its state approaches the

optimum solution with time [140]. Simulated annealing based methods have been found useful for accelerating NP-hard problems when implemented on massively parallel Boltzmann machines [19].

1.1 Focus and outline of thesis

In this thesis, we demonstrate the potential that emerging device technologies, specifically the spintronic device called the Magnetic Tunnel Junction (the central component of Magnetic RAM memory technology), possess for overcoming some of hurdles faced by modern computing systems. We show what role it can play in realizing hardware and accelerating computations performed in Artificial Neural Networks (NN) and combinatorial optimization through non-conventional paradigms and non-von Neumann architectures. The rest of the thesis is organized as follows.

- Chapter 2 provides background on the main concepts required to understand the contributions made in this thesis. These include the basic structure and workings of ANNs, spintronic and memristive devices, the rise of non-von Neumann computing platforms and imprecise computing models.

- Chapter 3 proposes an energy-efficient way of using MTJs for realizing NNs in a non-conventional domain called Stochastic Computing. We suggests ways of achieving energy-efficiency through NN parameter approximation and develop optimization algorithms for the same [88, 90].

- In chapter 4, we consider an MTJ crossbar based architecture for implementing NNs in a non-von Neumann manner. We discuss the drawbacks of directly programming the crossbar and propose methods for on-chip training of cross-

bars with different kinds of selectivity [89, 91].

- NP-hard problems are tackled in chapter 5 where we focus on a model that encodes such problems and uses simulated annealing to obtain good local optima. We propose a reconfigurable and parallel MTJ-based architecture which realizes the hardware representing the model and finds close-to-optimum solutions of the encoded problem. [92].

Chapter 2: Preliminaries

This chapter provides the background on several concepts which are the focus of this thesis. We start with describing the functioning and training of neural networks, move on to emerging non-CMOS devices with emphasis on spintronics, discuss the significance of non-von Neumann computing and the basic architecture for in-memory computing, and finally mention various forms of imprecise computing. Some specific topics that have been referred to only in a single chapter are explained there itself.

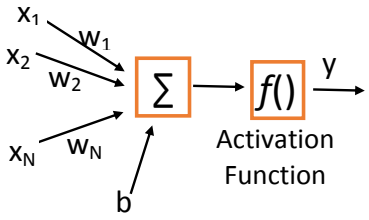
2.1 Artificial Neural Network Architecture

The fundamental units of a Neural Network (NN) are *neurons*, which represent non-linear, bounded functions, and *synapses*, which are interconnections between neurons. Each neuron performs a weighted sum of its inputs, which in turn is fed to a non-linear activation function to squash the output to a finite range [109]. The output of a neuron, called the *activation level*, can be expressed as

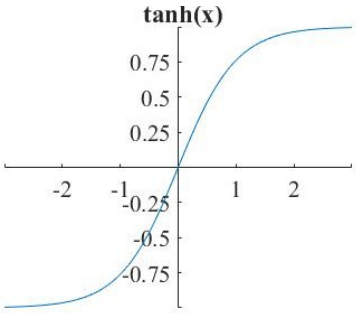
$$y = f\left(\sum_{i=1}^{N} w_i x_i + b\right) \tag{2.1}$$

where N is the no. of inputs to the neuron, w_i is the synaptic weight of the connection from the i^{th} input x_i, b is a bias, and $f()$ is an activation function (such as *tanh*

or ReLU). Fig. 2.1(a) depicts the operations performed by a neuron and 2.1(b), the behavior of the *tanh* function. A layer of neurons in an NN typically refers to a set of neurons which are not connected to one another, meaning that there are no synapses between these neurons (more biologically-inspired NN models can be exceptions). And a layer of weights is the set of incoming synaptic weights to that layer of neurons.



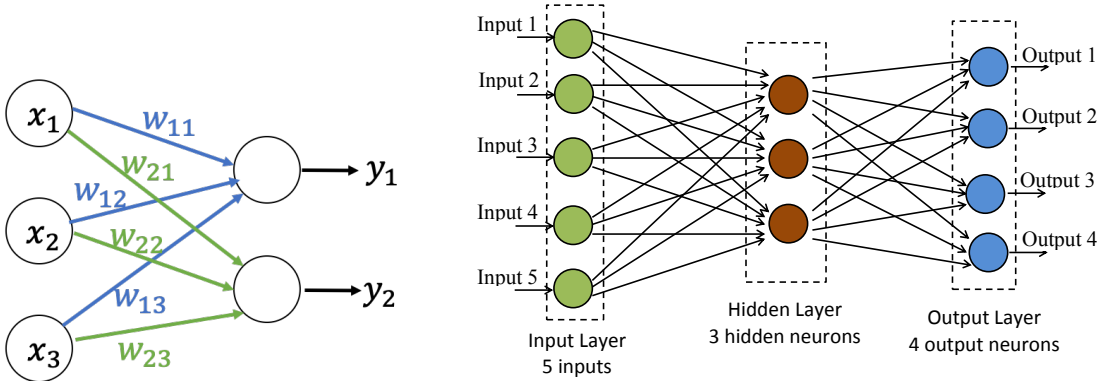
(a) A neuron perform a weighted sum of inputs and passes it through an activation function



(b) The transfer function of the *tanh*.

Figure 2.1: (a) Schematic of a neuron. (b) The *tanh* function.

Feedforward networks are the most elementary Neural Networks, in which information flows only in one direction from the input to the output, represented by an acyclic graph. The simplest of such networks is the fully connected one, which has connections from every input to every output neuron. Fig 2.2(a) shows a 3-input 2-output fully connected NN layer. For such a layer which is fully connected, its weights can be represented as a matrix W, and its output to any input vector x is given as $y = Wx$. This is known as forward propagation or *inference*. Multiple such layers can be connected in series (cascaded) to form the entire network, in which all intermediate layers are called *hidden layers*. Fig. 2.2(b) depicts a 2-layer NN with 3 hidden neurons.



(a) A fully connected layer of NN **(b)** Schematic of a 2-layer NN with one hidden layer.

Figure 2.2: Single-layer and 2-layer NNs

One very popular type of NN layer, in terms of connectivity are convolutional layers. Here, for each output neuron, weights exist only from a small set of the inputs which are located within its proximity. See [24] for more details on Convolutional Neural Networks. When several layers, convolutional or fully connected, are connected back to back, it forms what is popularly known as a Deep Neural Network (DNN). A DNN used for typical ML-based applications can have thousands of neurons and weights, and this is what makes their hardware implementation both computation and memory-intensive [83].

2.1.1 Training of Neural Networks

The ability of an NN to learn is what makes it useful. Prior to using in applications such as function approximation and classification, an NN has to be trained using several examples from a dataset, which are pairs of training inputs and their corresponding outputs or labels. The weights are initialized to random values and then adjusted as the network is trained to perform a certain task. The inputs in the training dataset are scanned one by one (often in batches). One single pass/iteration through the entire training dataset is called an *epoch*.

Training of the NN involves gradually adjusting the weight matrix W (or matrices) such that its output y moves closer to the target output t (for input x) at every step of the training. Towards this, a cost function is used to measure the deviation between the desired and the obtained output. One common cost function is the Mean Square Error (MSE) given as $E = ||y - t||_2^2$. The most popular technique of training an NN is the *error backpropagation* method, which relates the error or cost function with the weights of all the layers. This kind of a "backward calculation" is used to compute the gradient of the error function that is then used to update the weights in the direction in which error goes down the steepest [109]. This is known as gradient descent or the delta rule which is mathematically stated next.

Let the input to a single layer NN be $x \in R^M$, and $W \in R^{N \times M}$ represent the synaptic weight matrix, then the output $y \in R^N$ is

$$y = f(Wx) \tag{2.2}$$

where $f(\ )$ is an activation function. The weight update of the synapse connecting the i^{th} input to the j^{th} output is given as

$$\Delta W_{ji} = -\eta \frac{\partial E}{\partial W_{ji}} = -\eta x_i \delta_j \tag{2.3}$$

where E is the cost function of the presented input sample x, η is the learning rate and δ_j is the error calculated at the j^{th} output using y and the desired output t. For the single layer NN (or the last layer of a multi-layered NN), δ is directly

proportional to $(y - t)$. For hidden layers, error δ is obtained by backpropagating the error of the next layer. Thus, errors are computed starting from the last layer and ending at the first. The weight update of the entire matrix is the following outer product

$$\Delta W = -\eta \delta x^T \tag{2.4}$$

The new weight matrix is given as $W = W + \Delta W$. This weight update can be done after each training input is scanned or after accumulating the outer products from multiple inputs. See [70] for a thorough discussion on backpropagation and the various tricks that can be used to improve convergence of weights during training.

2.2 Spintronics and the Magnetic Tunnel Junction

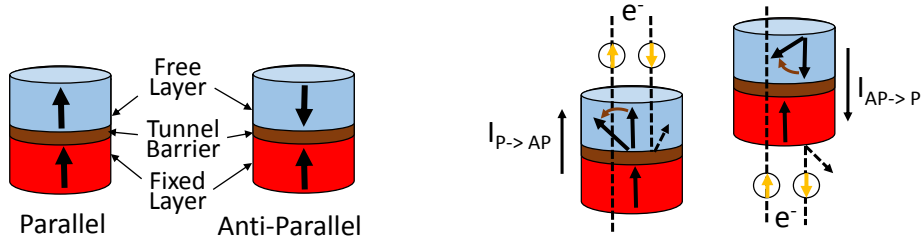
The CMOS technology is approaching the physical limits of scaling, which is giving rise to issues such as large leakage currents and high power dissipation density. This has fueled the search for alternatives to the CMOS technology for memory and logic [114]. Among all post-CMOS devices that are candidates for replacement, spintronic devices are one of the most promising ones [61]. Spintronics encompasses the field of magnetic electronics [15, 81] and refers to the use of electron spin for computation or storage. Unique features of such devices include non-volatility, zero leakage power, CMOS compatibility, etc. These characteristics have also enabled the implementation of new classes of architectures and inspired the development of novel algorithms suited to them [130]. The most popular and commercialized spintronics-based product is perhaps the Magnetic RAM (MRAM) which is starting to replace CMOS-based main memory and caches [131, 107]. The goal of semiconductor com-

panies is to establish a universal memory that can replace the mainstream ones by surpassing them in several criteria.

2.2.1 Magnetic Tunnel Junction

The central component of the MRAM is the Magnetic Tunnel Junction (MTJ). It is a 2-terminal spintronic device consisting primarily of 2 ferromagnetic layers separated by a thin tunnel barrier (typically MgO) [132]. The magnetic orientation of one of the magnetic layers is fixed, whereas that of the other is free, as shown in fig. 2.3(a). MTJs possess 2 stable states where the relative magnetic orientations of the *free* and *fixed* layers are Parallel (P) and Anti-Parallel (AP) respectively, with the P state exhibiting a lower resistance than the AP state ($R_P < R_{AP}$). It is this difference in resistance that allows a single-bit value to be encoded in the MTJ and which is characterized by the Tunnel Magneto-Resistance, $TMR = (R_{AP} - R_P)/R_P$.

The magnetization dynamics of the MTJ is governed by the stochastic Landau-Lifshitz-Gilbert (LLG) equation [77, 117]. It is possible to switch the state of the MTJ by passing spin-polarized current of appropriate polarity which flips the magnetization of the free layer through the mechanism of Spin-Transfer Torque (STT)



(a) Two stable states of an MTJ

(b) STT-assisted switching in MTJ

Figure 2.3: (a) The MTJ (b) Spin-Torque Transfer switching from P→AP (left) and AP→P (right). Dashed lines show the path of oppositely spin-polarized electrons.

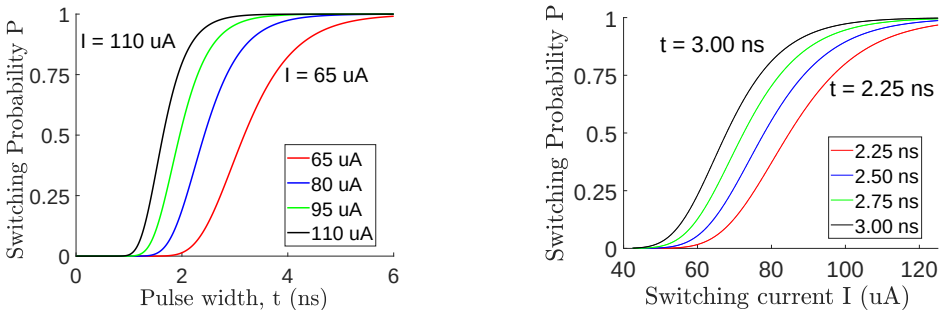
[77] (depicted in fig. 2.3(b)). The time required to switch is heavily dependent on the magnitude of the switching current. Not only that, this switching process is a stochastic one, in the sense that a pulse of given amplitude and duration has only a certain probability to successfully change the state. This stochasticity is due to thermal fluctuations in the initial magnetization angle and is an intrinsic property of the STT switching [77].

Depending on the magnitude I of the current and the critical current I_{c0} [142], the switching probability in the high-speed precessional regime ($I > I_{c0}$) is expressed as

$$P(a,t) = exp(-4f(a)\Delta exp(-2t/T)), \text{ with } f(a) = \left(\frac{2a}{a-1}\right)^{\left(\frac{-2}{a+1}\right)} \tag{2.5}$$

where $a = I/I_{c0}$, t is the pulse width, Δ is the thermal stability and T is the mean switching time (which is dependent on a)[127]. It must be mentioned that quantities such as Δ & I_{c0} and MTJ switching properties depend on device dimensions and material.

The spin transfer efficiency (θ) of an MTJ is a measure of how effectively charge currents are converted to spin-polarized currents. This θ is different for the 2 switching directions, with $\theta^{P\rightarrow AP}$ having a smaller value than $\theta^{AP\rightarrow P}$ [143].



(a) P v/s t, for different values of I **(b)** P v/s I, for different values of t

Figure 2.4: MTJ $AP \rightarrow P$ switching probability as a function of t and I

This makes $I_{c0}^{P\rightarrow AP} > I_{c0}^{AP\rightarrow P}$, which means that the same magnitude and duration of current will correspond to different switching probabilities for the 2 switching directions. Fig. 2.4 shows the dependence of the switching probability on pulse width and switching current for the $AP \rightarrow P$ transition. Observe the similarity in the nature of variation with I and t. The $P \rightarrow AP$ transition too depicts this kind of a behavior, albeit with different values of I and t.

2.2.2 Other spintronic devices

Other than the STT-MRAM, which has been used as memory, there exists spintronic devices for performing logic operations [61] such as

1. Hybrid MTJ/CMOS logic [124] - A pair of MTJs and a few transistors can implement most of the logic gates. This is one way of realizing logic-in-memory (see sec. 2.3).

2. All Spin Logic (ASL) devices - These consist of input and output magnets with a conducting channel in between, and utilize spin injection, spin diffusion and STT switching.

3. Domain Wall Logic - The domain wall is the interface between 2 magnetic domains; its motion can be used for logic operations.

4. Nanomagnet Logic - Utilizes magnetic direction as a state variable

2.2.3 Memristive devices

Another class of beyond-CMOS device which has caught the attention of researchers is the memristor, which has long been considered as the fourth basic circuit element

[25]. It is a resistive device which possesses a memory-like effect and a variety of dynamic characteristics. The fundamental physics of these devices differs from spintronic ones in the sense that their resistance depends on the ionic configuration of the material and the presence/absence of a conductive filament [144].

A common examples of a memristive device is the Resistive RAM (RRAM). It offers high integration density, non-volatility, and low-cost fabrication, and its resistance can be (re)configured through electrical inputs. Not only does it have a high and low resistance state, but some works have reported the existence of multiple intermediate resistance states [63]. RRAMs too have the potential for being used as memory [34, 54] and also for non-conventional computing as described next. See [23] for a detailed comparison between different emerging non-volatile memory technologies.

Although memristors tend to have intermediate resistances, it is often difficult to control their final state due to their highly non-linear behavior [96, 63]. Process variations and the resulting non-ideal device behavior make this worse. Thus it is difficult to obtain the intermediate states reliably. Further, there exists an asymmetry in the On-to-Off and Off-to-On switching [144].

2.3 Non-conventional and Non-von Neumann Computing

Another impact of the apparent end of Moore's law has been the birth of several new computing models that depart from the traditional and general-purpose ones [114]. Increasing the density of integration on chips will require lesser energy costs of data movement which depends on the intrinsic resistance of interconnect. As a result, computing efficiency is becoming increasingly limited by memory bandwidth,

which lags far behind processor computing speeds. In other words, memory has not scaled as much as logic, and therefore movement of data now constitutes a significant portion of energy consumption. For example, in data-intensive applications, off-chip memory access can account up to 90% of the execution time and energy [125]. Non von-Neumann computing seeks to bridge the gap between the processor and the memory by bringing them as close as possible or using the same physical entity for them.

The fundamental concept of tailoring the computing architecture to the needs of the application and nature of computation has been in use for a while in the form of ASICs, FPGAs, GPUs and GPGPUs, etc [125]. A relatively recent effort to solve the memory bottleneck includes bringing memory closer to processors through concepts such as logic-in-memory or memory-in-logic. Other similar methods include in-package memory, enhanced DRAMs and the 3D Crosspoint technology [53] which involve 3D integration and stacking. Although end users have seen improvements in energy-efficiency and performance as memory and processors are integrated closely, these gains wouldn't continue for long with the current memory devices and architectures.

A radical departure from von Neumann architectures involves in-memory computing, which essentially refers to doing computations right at the location of the memory. This solves the memory bottleneck by not requiring to fetch data from memory to the processor and writing data back to memory. The thrust in research in this direction has multiple sources:

1. The ever-increasing use of deep learning algorithms, which are often memory-intensive, for commercial workloads. Modern networks typically have tens of

thousands of parameters which require large amounts of storage, and hence large traffic from off-chip storage to on-chip processor.

2. The gap between the computational capabilities of CPUs and the human brain (with same amount of resources or power for fair comparison) has inspired researchers to better understand the working of the brain. Such brain-inspired computation requires special architectures which can offer very high levels of parallelism. Few examples of large-scale neuromorphic processors include the Stanford Neurogrid, Manchester SpiNNaker, Hiedelberg BrainScaleS machine and IBM TrueNorth which strike a balance between various performance objectives, and which differ in modes of operation (analog v/s digital) and neuron & synapse models [35].

3. The emergence of non-CMOS devices with unique characteristics, such as non-volatility and the ability to represent and process non-binary data. These devices can form crucial elements of non-von Neumann frameworks which can enable better realization of deep learning algorithms or other memory-intensive applications.

4. One important property of neural algorithms and their applications is their resilience to small errors in the input or the computations. Certain non-ideal characteristics of non-CMOS devices such as stochasticity not only have an insignificant effect on the result but also are desirable sometimes during training and operation [116, 94]. After all, biological NNs too function and learn with some uncertainty [121].

The most common form of non-von Neumann computing enabled by emerging

devices is analog computing, which involves computing with non-binary values by utilizing physical principles [114, 55]. An input signal in the form of a voltage, when provided to a resistive device, produces an output current that depends on its resistance and the voltage by Ohm's law. And several such resistors, when connected in parallel with a common output, would have their respective currents added up by Kirchhoff's law. This forms the basis of the in-memory compute capability of resistive devices and has the potential to realize the most fundamental computation of neural workloads and beyond. Next we discuss the most basic form of in-memory analog computing and its architecture.

2.3.1 The Resistive Crossbar Architecture

The mesh-like crossbar has been a popular architecture for memory. It's structure is suitable for performing matrix vector multiplications in the analog domain for neural or other applications. Often, inputs are provided to one side (say the rows) of the crossbars and outputs are obtained from the other side (the columns). Fig 2.5 shows a simplified crossbar (without access transistors) with M rows and N columns. For realizing an $M \times N$ NN weight layer, each row can correspond to an input and each column to an output neuron. Each resistive device at the junction of a row and column represents a synapse, whose weight would be related to the conductance.

The crossbar performs a mat-vec multiplication as a read operation in the following way. Let V_i be the voltage applied at the i^{th} input terminal and G_{ji} be the conductance of the synapse connecting it to the j^{th} output. By Ohm's Law, the current through that synapse is $G_{ji}V_i$ and by Kirchhoff's law the total current at

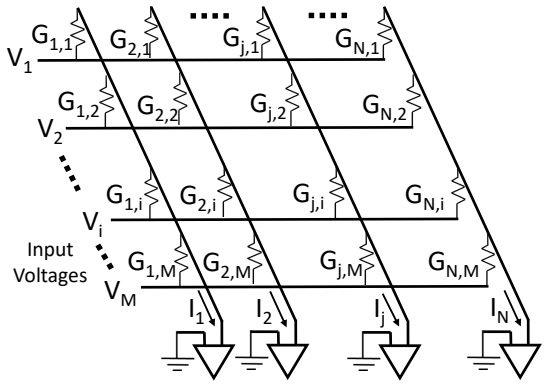


Figure 2.5: Structure of an $M \times N$ resistive crossbar

the output is

$$I_j = \sum_i G_{ji} V_i \tag{2.6}$$

which bears similarity to the dot product operation. This can then be either converted to a digital value with an ADC or fed directly to suitable analog circuits for implementing the activation function of the neural network [59, 47].

Since all M inputs can be applied simultaneously and the N outputs are obtained almost instantaneously, the matrix-vector multiplication is performed in parallel with constant time complexity. Whereas using conventional Multiply-and-Accumulate (MAC) units, the same could take up to $O(M \times N)$ time.

Memristive devices have also been proposed for performing logic operations in a variety of ways. Several logic design styles exist where the input(s) or the output can be represented with voltages or resistances [33]. The work in [123] realizes Boolean functions (AND, NOR, etc.) using stateful logic within memristive crossbars and demonstrates the operation of full adders. Use of spintronic devices for logic gates and circuits have also been proposed [56, 32].

Fig. 2.6 presents new devices, computing architectures and paradigms along different dimensions which can tackle the challenges faced by classical computing

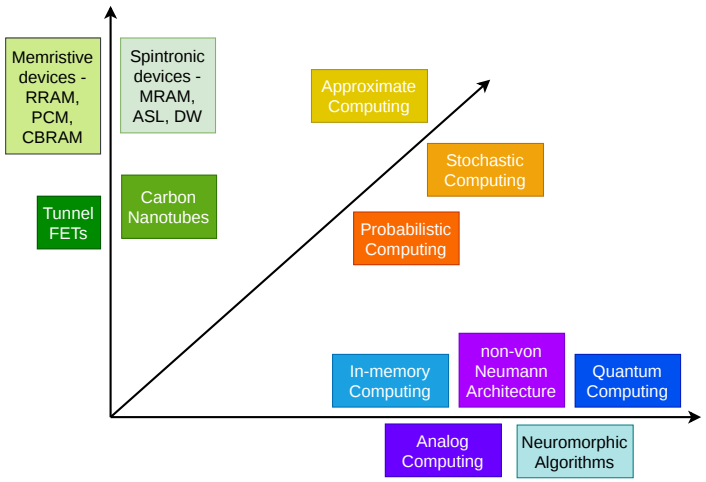


Figure 2.6: Opportunities to overcome the hurdles presented by the slowing down of Moore's law as 3 paths, not necessarily meant to be used mutually exclusively.

with CMOS technology.

2.4 Non-conventional computing paradigms

The ever-increasing amount of data that is processed by modern computers has led to a sharp rise in power consumption in spite of technological advancements. On the other hand, the shrinking of transistors has increased the chances of device failure and transient errors. This has given birth to the concept of imprecise computing which advocates tolerating some errors in the computation for achieving lower power or design area. A growing number of applications handled are resilient to small errors or noise in the data, algorithms, and circuits. The notion of imprecise computing has found use particularly in Machine Learning and Big Data applications, where one or more of the following hold [45]

- There doesn't exist a single correct/golden answer
- Obtaining the correct answer takes up a lot of time and energy
- Any approximate answer is as good as the correct answer due to limitations in

human perception and/or error tolerance of the application.

We shall now discuss few categories of imprecise computing, which, it must be mentioned, are not always mutually exclusive.

2.4.1 Approximate Computing

This is most common form of imprecise computing which trades-off accuracy of results for lower energy and area [137]. The simplest example is using simplified logic for obtaining the less significant bits of a computation so that the errors in the result are within an acceptable level. Of course, approximate computing is to be only employed in non-safety critical domains.

2.4.2 Probabilistic Computing

It refers to computing results with less than 100% probability of correctness by, for example, using computing elements that have higher than usual levels of noise. With CMOS gates, an external source of noise may be used, whereas non-CMOS logic may have inherent randomness [104, 61]. While probabilistic computing also produces answers which are approximately correct (or at least desired to be so), there is non-determinism involved in it. Whereas the term approximate computing usually refers to deterministic approximations in computing logic and data.

2.4.3 Stochastic Computing

Stochastic Computing (SC) specifically refers to the use of bitstreams for representing data and using simple logic gates for arithmetic functions [12]. Herein, the data

is approximate and processed serially but the computations are generally exact. SC drastically reduces area and power consumption, while occasionally increasing the latency of computations. Another challenge that SC faces is bitstream correlation that tends to reduce the accuracy of results. More technical details of SC will be discussed in the next chapter.

The above forms of imprecise computing may or may not be used in the context of non-von Neumann computing. Computing systems in the future are likely to be heterogeneous in the sense that they would use a blend of different computing paradigms, each suited to a particular set/type of applications, realized with hybrid CMOS/non-CMOS technologies.

Chapter 3: Stochastic Computing with MTJ for Neural Networks

In this chapter, we consider the union of Stochastic Computing and spintronics for realizing a Neural Network (NN) architecture and optimizing it for energy-efficiency.

3.1 Introduction

In the previous chapter, we discussed the fundamentals of the workings of an NN and noticed that it primarily comprises a large number of multiplications and additions (MAC) which can be done in parallel for any layer of the NN. Although NN applications have been run on GPUs, FPGAs and high-performance servers to take advantage of parallelism, such designs with binary MAC units would have a high cost in terms of area and power consumption. This characteristic prohibits their deployment in embedded and IoT devices, where both of those metrics are desired to be low. It has prompted the development of optimization techniques at different levels of these complex networks to achieve energy efficiency [128, 93], and has also motivated the use of computational paradigms different from the traditional ones.

Stochastic Computing (SC) is a great candidate for replacing the conventional multipliers and adders of an NN. Its use of bitstreams to represent data enables the use of simple logic gates for arithmetic operations. Further, the inherent error-resilience of Recognition, Mining and Synthesis applications easily allows for the

errors produced in data due to SC. However, data in SC, called Stochastic Numbers (SNs), are generated by circuits called Stochastic Number Generators (SNGs). Traditionally, SNGs are composed of pseudo-random number generators (such as Linear Feedback Shift Registers - LFSR) and comparators [12], which can account for a significant fraction of the design cost of the complete system in terms of energy and area. For eg. the SNG's energy consumption can be up to 80% when implemented using LFSRs [11, 129]. Thus, designing low-cost SNGs is of prime importance to the overall energy-efficiency of SC-based circuits.

In this chapter, we integrate SC based on MTJs into ANNs and explore the different ways of achieving energy efficiency at both the device level and the network level, in the latter through approximations. Our contributions are summarized as follows:

- We outline the characteristics of an MTJ with regard to switching time and energy, develop a low-energy MTJ-SNG by exploiting the properties of SC, and compare it with the baseline.
- We propose the use of our MTJ-SNG as an architectural construct for ANNs in the SC domain, and develop an optimization algorithm that approximates the synaptic weights in a single-layer NN for achieving energy-efficiency by sacrificing little accuracy.
- This algorithm is then extended to a multi-layer NN by heuristically breaking down the entire problem into separate problems for each layer and solving each of them optimally.
- Lastly, we show how regularization techniques can be incorporated in the NN training process to obtain better results, and prove the effectiveness of our algo-

rithm through simulations.

3.2 Preliminaries

3.2.1 Stochastic Computing

The concept of Stochastic Computing (SC) and other closely related computational paradigms dates back to the 1960s and 70s [36, 100, 101], and essentially refers to the representation of analog quantities by probabilities of discrete events which occur sequentially and are statistically independent. In contrast to conventional arithmetic computing, SC uses bit streams to represent numbers, typically denoted by the probability of '1's in the stream. A Stochastic Number (SN) with value $p \in [0, 1]$ is represented as a *Bernoulli sequence* of bits, such that if there are n bits in the sequence, out of which k are '1', then $p = \frac{k}{n}$ [11]. This is known as the *unipolar* format. In the *bipolar* format, $p \in [-1, 1]$, and the same bit sequence would now have the value $p = \frac{2k-n}{n}$. For example, the bit stream 0100101000 would be interpreted as 0.3 in the unipolar format and -0.4 in the bipolar format.

Fig. 3.1 shows the hardware components required for SC. Traditionally, the SNG comprises the LFSR whose output is compared with the binary representation of the SN desired to be generated. The SNG's output is used by circuits described next, and the final result can be converted back to the binary format with a counter. In this thesis chapter, the main focus in terms of techniques proposed would be on the SNG.

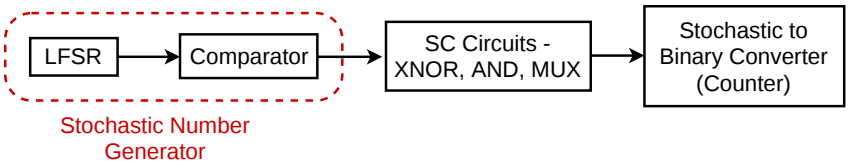
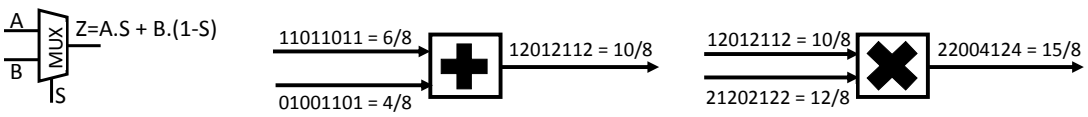


Figure 3.1: Components used in SC and direction of flow of data. The LFSR and the Comparator make up the SNG.

3.2.2 Computational units in SC

In SC, multiplication is performed by an AND gate in the unipolar format [11]. Thus, given 2 stochastic streams X and Y, their product is AND(X,Y). In the bipolar format, it is given as XNOR(X,Y). However, it is not possible to perform a precise addition in the SC domain as the sum of 2 SNs might very well lie beyond the range. Only a scaled addition is possible which is achieved through a 2:1 Mux whose Select input is the scaling factor and is also an SN. The scaled addition of A and B, with scaling factor S, would give Z = A.S + B.(1-S) as in fig. 3.2(a). With $S = 0.5$, one can get $\frac{A+B}{2}$, albeit with a loss of precision. However, most implementations of NNs involve the sum of a large number of numbers and a loss of precision would only result in severe errors at its outputs.

To overcome this issue, Ardakani et al. [14] introduce the concept of Integral Stochastic Computing (ISC) which allows us to represent numbers beyond the range of conventional SC. In the unipolar format, a real number $s \in [0, m]$ can be expressed as the sum of m numbers $s_1, s_2, ...s_m \in [0, 1]$. Each s_i can be represented



(a) **(b)** $0.75 + 0.5 = 1.25$ **(c)** $1.25 \times 1.5 = 1.875$

Figure 3.2: (a) Scaled addition in SC, (b) Integral SC (ISC) representation ($m = 2$), and (c) Multiplication in ISC ($m_1 = m_2 = 2$)

as stochastic streams and s can be obtained as the bit-wise summation of these m streams as illustrated by an example in fig. 3.2(b). For eg., 1.25 can be expressed as $0.75 + 0.5$ which have 8-bit stochastic representations (say) 11011011 and 01001101 respectively. Now, the integral stochastic stream of 1.25 can be obtained by a bit-wise summation of these, which is 12012112, also represented using 2 streams.

In general, a number $s \in [0, m]$, when represented as the sum of m SNs, would require $\lceil log_2 m \rceil + 1$ streams (similar to a binary representation). This concept extends similarly to the bipolar format as well [14]. Multiplication and addition in ISC are performed using binary radix multipliers and adders respectively. Given 2 real numbers $s_1 \in [0, m_1]$ and $s_2 \in [0, m_2]$, their product and sum would have $\lceil log_2(m_1 m_2) \rceil + 1$ and $\lceil log_2(m_1 + m_2) \rceil + 1$ bits respectively in the ISC domain. Fig. 3.2(c) gives an example. It must be noted that though computations in ISC require binary radix adders and multipliers, these are much less expensive than those in conventional methods of computing. For example, addition of two integral SNs with $m_1 = m_2 = 2$ and precision $1/n$, will need a 2-bit adder irrespective of their precision; whereas the same in arithmetic computing will need a $(1 + log_2\ n)$-bit adder. The difference is same for the case of multiplication.

3.3 MTJ-based Stochastic Computing

In this section we shall describe the characteristics of an MTJ with regard to its probabilistic switching and exploit the properties of Stochastic Numbers to design a low-energy optimized MTJ-based SNG and compare it to its non-optimized version. This MTJ-SNG would be the underlying source of approximations in our energy-efficient NN implementation.

3.3.1 Characteristics of Magnetic Tunnel Junctions

Recall from chapter 2 that the MTJ has 2 stable states with Parallel (P) and Anti-Parallel (AP) magnetizations. And that the state can be switched by passing a current through it, although such a switching is probabilistic in nature (see fig. 2.4). Thus, a higher switching current magnitude or pulse duration is required for a higher switching probability.

Let us now analyze theoretically the switching time and energy consumption of the MTJ. Let I_{AP} and I_P denote the currents for the $AP \rightarrow P$ and $P \rightarrow AP$ switching respectively. Given a pulse of width T_p, the expected time at which switching takes place (given it does) is expressed as

$$t_{sw} = \int_0^{T_p} \tau \frac{dP}{dt} d\tau \tag{3.1}$$

where the derivative of P is the switching probability density function with respect to time t (fig. 2.4a) for currents I_{AP} or I_P. The expected energy consumed in such a scenario, for AP→P switching, is

$$E_{sw}^{AP \rightarrow P} = V \left(I_{AP} t_{sw} + I_P (T_p - t_{sw}) \right) \tag{3.2}$$

where V is the applied voltage bias. Whereas the energy spent in the case where switching does not take place is

$$E_{nsw}^{AP \rightarrow P} = V I_{AP} T_p \tag{3.3}$$

The expected energy consumed is therefore given as

$$\langle E \rangle^{AP \to P} = P(T_p) E_{sw}^{AP \to P} + (1 - P(T_p)) E_{nsw}^{AP \to P} \tag{3.4}$$

Expressions are similar for the P→AP switching.

3.3.2 MTJ as a Stochastic Number Generator

Prior works [31, 135] have suggested the use of MTJs as an SNG by exploiting the probabilistic nature of its switching. We propose an energy-efficient version of an MTJ-SNG that is based on the same principles, but takes advantage of a trivial property of SC to achieve significant energy gains.

Given a voltage pulse, the probability of switching can be decided by controlling the pulse width. For each bit generated by the MTJ representing a stochastic number $p \in [0, 1]$, one would typically do the following iteratively:

i. Reset to '0' with 100% probability (not required if state didn't change in the previous iteration)

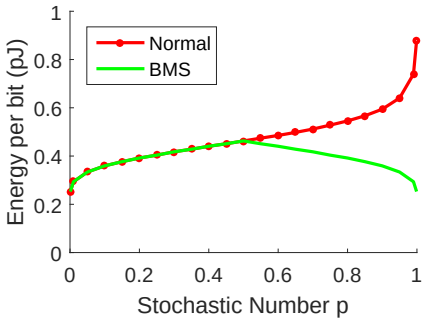
ii. Write '1' with probability p, and

iii. Read the value stored in the MTJ (which would be '1' with probability p and '0' with probability $1 - p$).

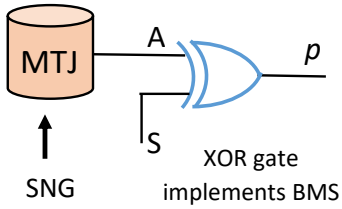
Repeating this procedure n times would give us a sequence of n bits, out of which $p.n$ are expected to be 1, thereby representing the SN p.

We thus choose the P state to be the reset state (logic 0), and switch to the AP state (logic 1) with some probability for generating the SN. This means that switching P→AP with probability p will produce bit streams where the probability

of finding '1's is p. The red dotted line in fig. 3.3(a) plots the relation between the expected energy consumption and switching probability at this bias voltage, which has been obtained with the help of an HSPICE model [60] of MTJ .



(a) Energy v/s probability for P→AP, $V_{bias} = 1.0V$



(b) Circuit of the BMS

Figure 3.3: (a) Variation of energy with value of SN p with and without BMS (green undotted and red dotted lines respectively). (b) The BMS.

3.3.3 Proposed Biased MTJ-SNG

We make a slight modification to the overall procedure of generating the bits of the SN. As seen earlier, if p is closer to 1 than to 0, more time, and hence more energy, has to be spent in writing '1' to the MTJ, as compared to the case where we had to generate an SN with value $1 - p$.

To reduce the average energy of the MTJ-SNG, we choose to generate $1 - p$ whenever $p > 0.5$ (but generate p if $p \leq 0.5$). In other words, whenever $p > 0.5$, instead of switching P→AP with probability p, we switch with probability $1 - p$ (which is ≤ 0.5) and invert the bits output from this Biased MTJ-SNG (**BMS**, the name being derived from the biased nature of the data produced by the MTJ-SNG) so that we get back the SN p. Therefore, we generate either p or $1 - p$, whichever is smaller, and use an XOR gate to choose between the generated SN and its inverse

as shown in fig. 3.3(b). The 'S' input can be derived from the most significant bit of the binary number that is being converted to a stochastic number [11]. As an example, if $p = 0.3$, the MTJ-SNG will generate p itself and S will be 0 to output $A = 0.3$. On the other hand, if $p = 0.7$, the MTJ will generate $(1 - p)(= 0.3)$ and S will be 1 to output $\bar{A} = 0.7$.

The energy required to generate one bit from the BMS is plotted (green undotted line) in fig. 3.3(a) as a function of p. The symmetry of the plot comes from generating the smaller of p and $1 - p$. Table 3.1 compares the 2 MTJ-SNGs in terms of the total time, average energy, and average power required per bit output. The XOR in the BMS has a small contribution of $0.1\mu W$. Since the BMS requires us to generate SNs only lesser than or equal to 0.5, the maximum write duration reduces from $5.46ns$ to $2.34ns$ (the latter corresponds to the pulse width giving 50% switching probability), thereby decreasing the total time. The average energy and power have been calculated considering a uniform distribution of p over the range [0, 1]; BMS brings about a reduction by 27.5% in energy (without introducing any approximation or error in the SN being generated). The power doesn't scale with the energy as the write latency also reduces.

Table 3.1: Comparison of Normal and Biased MTJ-SNG

MTJ-SNG	Time(ns)	Avg. Energy (pJ)	Avg. Power (μW)
Normal	11.33	0.726	64.08
BMS	8.21	0.526	64.07

3.3.4 Comparison with CMOS-based SNG

The authors in [129] report that a spintronic-based SNG built with the MTJ can be 7 times more power efficient than a CMOS-SNG. Knag et. al. [64] synthesize a 100

MHz SNG with a 32-bit LFSR and a comparator in $65nm$ technology, which has a power consumption of $80.2\mu W$. This translates to an energy consumption of $0.8pJ$ per bit of the SN having a throughput of 1 bit every $10s$. These figures are slightly worse than our BMS which produces a bit every $8.21ns$ with an energy of $0.53pJ$.

It is worth noting the following in terms of scalability and power of SNGs. The power of a CMOS-based SNG (LFSR + comparator) scales linearly with the size of the LFSR and the comparator, which strictly governs the precision of the SN generated. But an MTJ-based SNG would have a power consumption independent of the desired precision of SNs.

3.4 Energy Efficient MTJ-based NN Implementation

Stochastic circuits have gained popularity in low-cost implementation of NNs [14] [62]. We propose using MTJs as a hardware component for realizing NNs in the SC domain by exploiting their probabilistic switching nature to generate SNs representing inputs and synaptic weights. The error-resilient nature of NN applications motivate us to approximate the network outputs, and hence the weights, effectively designing approximate multipliers, and thereby gaining energy efficiency. In this section, we develop an algorithm that, given a trained network, the training dataset and an error tolerance, adjusts the weights in the best possible way in the solution space, while remaining within the error constraint at all times.

3.4.1 NN implementation in the SC/ISC domain

Here we describe how the operations of a neuron would be performed in the ISC domain (described in section 3.2.1). We know that the activation level of a neuron is given as

$$y = f(a) = f\left(\sum_{i=1}^{M} \tilde{w}_i \tilde{x}_i\right) \tag{3.5}$$

where f is the activation function operating on a, the weighted sum of inputs. Several types activation functions can be used in an NN; we go with the *tanh* function.

In eqn. (3.5), the $\tilde{x}_i$ (inputs) are assumed to be in the range $[-1, 1]$ (if not, they can be normalized). Let the $\tilde{w}_i$ (weights) be in $[-\beta, \beta]$. The latter can be represented in the ISC domain with $\lceil \log_2 \beta \rceil + 1$ stochastic streams. However, if $\beta > 1$, this would need those many SNGs, leading to higher area and energy consumption. On the other hand, if $\beta < 1$, producing SNs equal to the value of the weights would mean an under-utilization of the available range/precision. Therefore, we have to scale them down to the range $[0, 1]$ or $[-1, 1]$ to be able to use only 1 stream, and that too effectively. Since the ISC implementation of the *tanh* function using FSM [21, 14] is in bipolar format, we go for the interval $[-1, 1]$. So the weighted sum would now be written as

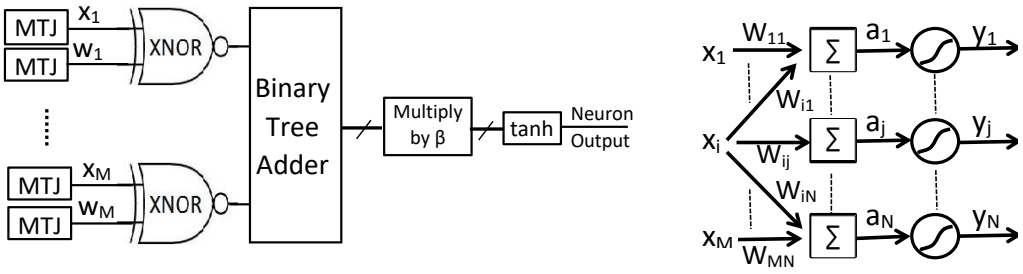
$$a = \beta \sum_{i=1}^{M} w_i x_i \tag{3.6}$$

where $x_i, w_i \in [-1, 1]\ \forall\ i$ and would be represented as stochastic numbers. Fig. 3.4(a) illustrates the operations of a neuron in the ISC domain, implementing eqns. (3.5) and (3.6). The addition, multiplication and neural activation would

be achieved as explained in sec. 3.2.1. Several such neurons in parallel would form a layer as in fig. 2.2(a), and multiple layers connected in series would make up the entire network. Note that the output of the *tanh* is a single stochastic stream in the bipolar format.

3.4.2 Problem Formulation

As can be seen from fig. 3.3(a), the generation of SNs (from the proposed BMS) that are closer to 0 or 1 require less energy as compared to those that are closer to 0.5. In the bipolar format of SC, this would imply low energy requirement for numbers closer to 1 or -1 than to 0. This property of the BMS forms the basis of achieving energy-efficiency through approximations that tend to shift the weights "farther from" 0 towards 1 or -1, whichever is closer. We therefore aim to bring the weights of the network as close to 1 or -1 as possible while ensuring that output errors are within a specified tolerance level for all the training inputs. We investigate both single-layer and multiple-layer NNs.



(a) **(b)**

Figure 3.4: (a) Neuron implementation in ISC. The outputs of the binary tree adder and multiplier consist of multiple bit-streams. (b) Schematic of 1-layer NN

3.4.3 Optimizing a 1-layer NN

For a single layer network, we illustrate how to formulate the network approximation as a convex optimization problem. Convexity of the feasible region of such a problem implies that any local minimum in that region is also the global minimum, ensuring that the optimum value of the objective function is always achieved. Further, non-convex optimization problems are more complicated to solve.

The objective of our formulation is to minimize the separation of the weights from 1 or -1 (whichever is closer). Since the weights are independent of each other, the objective function can be expressed as the sum of the "distance" of the weights from 1 or -1. One way of specifying an error tolerance at the output layer is to measure the deviation of the output neurons from their actual values (the values obtained from the trained network) and restrict all of them to within some threshold. Such a constraint should be applicable to all input vectors used in the optimization.

However, the *tanh* function (which provides the neuron output) is not only non-linear but also non-convex. Thus, neither neuron activation levels nor the errors in them can be directly incorporated in the convex formulation. But the input to this activation function is affine (hence convex) because it is a weighted sum of inputs. We therefore need to translate the output errors to errors in inputs of *tanh*. Given a limit to the deviation in neuron output, we pre-compute the upper and lower limits of the weighted sum input using the $tanh^{-1}$ function and force it to remain within these limits. Since *tanh* is a monotonically increasing (hence invertible) function, these limits can be computed exactly. Thus, the non-convexity of the *tanh* function neither impedes the optimization process nor introduces any inexactness.

Table 3.2: Notations for problem formulation of 1-layer NN

Name	Meaning	Type	Dimension
W	The output layer weights	Matrix	$M \times N$
$\hat{x}^r$	The r^{th} training input sample	Vector	M
β	The scaling factor for W	Scalar	1
a^r	The r^{th} weighted sums (output layer)	Vector	N
y^r	The r^{th} activation levels (output layer)	Vector	N

Fig. 3.4(b) illustrates a 1-layer network having M inputs and N outputs and table 3.2 lists the notations. In addition, the presence of $\hat{}$ (hat) symbol indicates that the quantity is the original value obtained from the trained network, and hence is a constant in the problem; whereas its absence denotes a variable.

The Optimization Procedure: The procedure for approximating weights in a 1-layer NN is shown below[1]. It takes a trained network and an error threshold ϕ as inputs, and minimizes the "sum of distances" using D samples of the training dataset. The Absolute Value (AV) of the deviation of the neuron output shouldn't exceed ϕ.

Line 2 computes the maximum and minimum values that the weighted sum inputs of the *tanh* function can take. Here y_j^r denotes the output of the j^{th} neuron for the r^{th} training input, and u_j^r & v_j^r are the corresponding limits. The objective function (line 3) to be minimized is the sum of distances of the weights from 1 or -1. W' in line 5 stores how far they are from 1 or -1, whichever is closer. It effectively implements $W'_{ij} = min(1 + W_{ij}, 1 - W_{ij})$; however this expression cannot be directly used as the minimum of affine functions is not convex [20]. This is also the reason why we impose a constraint on the range of the weights in line 4

[1]For solving the optimization problems we use CVX, a package for specifying and solving convex programs [42]. The way in which certain specifications (constraints and expressions) in the procedure are written is guided by the disciplined convex programming rules of CVX

Weight Approximation for a single-layer NN

1: **procedure** OPTIMWEIGHTS(M,N,$\hat{W}$, $\hat{x}$, $\hat{y}$, β,ϕ)
(In the following, i, j and r run from 1 to M, N and D respectively)

2: The constraint on the neuron outputs are $\left|y_j^r - \hat{y}_j^r\right| \leq \phi$.
Compute the upper and lower limits of all weighted sums as
$u_j^r = tanh^{-1}\left(\hat{y}_j^r + \phi\right)$ and $v_j^r = tanh^{-1}\left(\hat{y}_j^r - \phi\right)$ respectively

3: Solve the optimization problem: $\underset{W}{\textbf{minimize}}\ W_{sod} = \sum_{i=1}^{M}\sum_{j=1}^{N} W'_{ij}$
subject to the following constraints (lines 4 to 7):

4: Restrict the weights to their original range: **if** $\left(\hat{W}_{ij} \geq 0\right)$ **then** $0 \leq W_{ij} \leq 1$
else $-1 \leq W_{ij} \leq 0$

5: Find the distance of the weights from 1 or -1, whichever is closer

$$W'_{ij} = \begin{cases} 1 + W_{ij} & if\ \hat{W}_{ij} \leq 0, \quad (3.7) \\ 1 - W_{ij} & otherwise \quad (3.8) \end{cases}$$

6: Compute the weighted sum to all neurons for all inputs: $a^r = \beta(W^T \hat{x}^r)$

7: Constrain these weighted sums within their upper and lower limits: $v_j^r \leq a_j^r \leq u_j^r$

8: **return** $y^r = tanh(a^r)$

9: **end procedure**

(minimum of distance from 1 and -1 isn't convex). Line 6 computes the weighted sum inputs of the *tanh* function, line 7 constrains them within the limits obtained in line 2, and line 8 finally returns the approximate neuron outputs which can now be used to check the accuracy of the NN. The optimization problem stated above is convex because the objective function and the inequality constraints are convex and the equality constraints are affine [20].

3.4.4 Optimizing 2-layer NNs

A similar formulation could have been made for NNs containing more than 1 layer, having the objective of minimizing the "sum of distances" of each of the weight matrices, with constraints computing the hidden layer(s) outputs and finally re-

stricting the error in the output layer's weighted sums. However, the presence of the non-convex activation function in the hidden layer(s) would make the problem (as a whole) non-convex.

To mitigate this issue, we propose breaking down the problem into separate but identical convex problems, each of which optimizes the weights in successive layers of the NN under some error constraints. Thus, in a 2-layer NN having M inputs, L hidden neurons and N output neurons, we shall solve 2 problems successively - first for the hidden layer and then for the output layer, with error thresholds ϕ_Z and ϕ_W respectively. Given some value of ϕ_W, there exists an upper limit to the amount of error that can be tolerated at the outputs of the hidden layer which can be obtained using principles of linear algebra [88].

3.5 Regularization and Constraints for Classification problems

We now introduce 2 methods to improve the trade-off between energy and error rate of the MTJ-based NN implementation proposed in the previous section. These are - Regularization, to influence the distribution of the weights of the network in a way that leads to lower energy; and a modified way of specifying error constraints applicable to classification problems.

3.5.1 Regularization

This is a technique used primarily to prevent the over-fitting of networks on the training datasets. It is achieved by adding an extra term, known as the penalty function, to the cost function to be minimized during training. It has the effect of

changing the distribution of the weights of the network. With regularization, the overall loss function is expressed as

$$E = E_I + E_P(W) \tag{3.9}$$

where E_I is the error function (such as Mean Square Error or cross-entropy loss) computed from the inputs and the weights, and E_P is the regularization penalty function dependent solely on the weights. The weight update using gradient descent is now written as

$$\Delta w_i = -\eta \left(\frac{\partial E_I}{\partial w_i} + \frac{\partial E_P}{\partial w_i} \right) \tag{3.10}$$

Commonly used regularization functions are the $L1$ and $L2$ norms of the weights that impose a penalty on weights with a large magnitude and prevent them from growing by a large extent. However, the concept can be used in general to minimize any penalty function suited for the purpose. For eg. in [134], a wedge-shaped function is used to assist the network in learning discrete weights. Recall that the BMS consumes a lower energy when it has to produce SNs close to 1 and -1, which correspond to weight values β and $-\beta$ respectively. This preference for extreme values of the weights can be incorporated in the training of the network. We propose 3 kinds of regularization functions that push weight values to their extremes.

Type 1: A function that is maximum at 0 and keeps decreasing with increasing magnitude of the weights.

$$f(w) = \lambda \left(1 - \frac{|w|}{w_0} \right) \tag{3.11}$$

The derivative of this function is given as

$$f'(w) = -\frac{\lambda}{w_0} sign(w) \tag{3.12}$$

With this penalty function, the weights will always have a tendency to move away from 0. Note that it is only the ratio of λ and w_0 that affects the magnitude of the slope. Both equations have been graphically depicted below in fig. 3.5(a) and (d). So when $w > 0$, $f'(w) = -\lambda/w_0$ and $\Delta w > 0$, pushing the weight away from 0. However, one disadvantage of using this is that because it impacts all weight values equally irrespective of their magnitude, there is a high chance that the value of β would also go up.

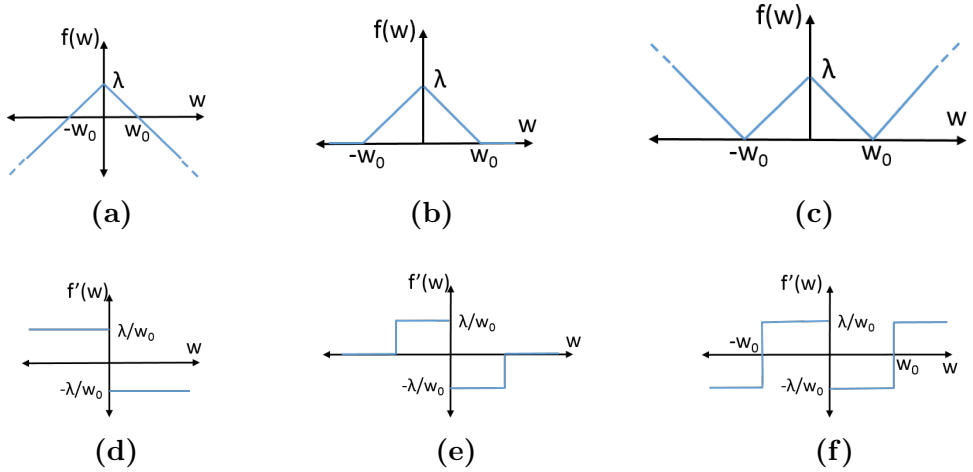


Figure 3.5: (a)-(c) Regularization function types 1,2 and 3 respectively and (d)-(f) their derivatives

Type 2: To counter the increase in β, we can impose a penalty on only those weights that are close to 0. Such a function can be defined as

$$f(w) = \begin{cases} \lambda \left(1 - \dfrac{|w|}{w_0}\right) & \text{for } -w_0 \leq w \leq w_0 \\ 0 & \text{elsewhere} \end{cases} \tag{3.13}$$

Thus, weights that are beyond the range $[-w_0, w_0]$ are not affected by the regularization as depicted in fig. 3.5(b) and (e).

Type 3: While everything is fine with type 2 regularization, it might be beneficial to attempt to reduce the value of β itself, while also keeping the weights away from 0. Such an objective can be achieved with

$$f(w) = \left| \lambda \left(\frac{|w|}{w_0} - 1 \right) \right| \tag{3.14}$$

This will try to bring the weights closer to a suitably chosen w_0 as plotted in fig. 3.5(c) and (f). In our experiments, w_0 was selected as the mean of absolute value of the weights obtained without regularization and the same was used for all 3 types of regularization. The reason behind this is that the mean value minimizes the $L2$-norm of its difference from the weights. The effect of the penalty function on the weight change Δw depends only on the derivative $f'(w)$, and can be adjusted by tuning the value of λ.

3.5.2 Classification Specific Customization

In section 3.4.3, we put a constraint on the Absolute Value (AV) of the error at each of the output neurons, which was then translated to upper and lower limits of the input of the *tanh* activation function. Classification problems typically have as many output neurons as the number of classes, and the one corresponding to the neuron having the highest value is taken as the output. That is

$$Class = k = \arg\max_j y_j \tag{3.15}$$

with y being the output from the last layer. As long as the k^{th} output remains the highest, the input will be classified to be of class k. This leads to a different formulation of the error constraint for such NNs where the k^{th} output is only allowed to increase and the rest can only decrease. Mathematically, this means

$$y_k \geq \hat{y}_k \;\; and \;\; y_j \leq \hat{y}_j \; \forall \; j \neq k \tag{3.16}$$

This is equivalent to having only a lower limit for the input of the k^{th} neuron and an upper limit for the others,

$$a_k \geq \hat{a}_k \;\; and \;\; a_j \leq \hat{a}_j \; \forall \; j \neq k \tag{3.17}$$

With some relaxation ϕ in the error of the neuron outputs, we may write

$$a_k \geq v_k \;\; and \;\; a_j \leq u_j \; \forall \; j \neq k \tag{3.18}$$

in line 7 of the optimization procedure, where v_k and u_j would be computed as in line 2. We shall, henceforth, refer to this modified error constraint by the name Classification Specific (CS). It is to be noted that the above constraints would be applicable only to the last layer of the NN; all hidden layer neurons would still have a restriction on the absolute value of the error as such strict ordering of output does not exist for them.

3.6 Simulation Methodology and Results

3.6.1 Evaluation setup

Several benchmarks based on classification problems were used to measure the performance of the NNs and estimate the energy savings obtained by approximating the multiplications. Training and optimization of the neural networks was done in MATLAB on a 64-bit computer with Intel Xeon E3 processor and 32 GB RAM. First, we train an NN in MATLAB with the mean square error cost function using the gradient descent method and check its accuracy on the test dataset. We then estimate its energy consumption in classifying one sample with the Biased MTJ-SNG (BMS), considering bitstreams of length 64. This energy includes those of the SNGs used for generating both the network inputs and the weights. The energy consumption of each BMS in the networks consists of 3 terms:

- The write energy, which varies with the SN being generated, and which is obtained from the data corresponding to the green plot in fig. 3.3(a)
- The read energy, and
- The expected energy required to reset, which again depends on the generated SN. Larger the SN, higher its chances of requiring a reset (although, recall that using BMS means we reset at most half of the times).

For the input BMS, we considered the average energy over all samples of the test dataset since different samples would have different energy requirements.

Next, we approximate the network using the optimization technique described in section 3.4 for different levels of error tolerance using CVX, a MATLAB-based

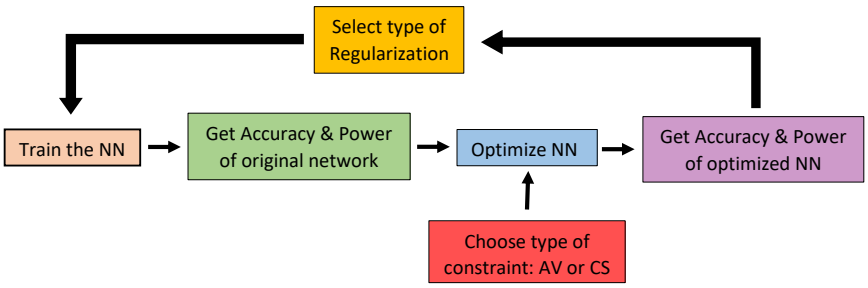


Figure 3.6: Flow chart showing the process and network optimization and characterization. We start with training an NN without regularization.

software tool for solving convex programs [42]. Finally, each of the newly obtained NNs with approximate multipliers were analyzed for their accuracy and their energy, again for bitstream length of 64. The input samples and weights of the networks were thus rounded off to account for the reduced precision. The entire process is repeated with the 3 types of regularization, and each of the 4 networks were optimized using both types of error constraints - Absolute Value (AV) and Classification Specific (CS). This is illustrated in fig. 3.6.

3.6.2 Results

The results from the different datasets are summarized below:

1. MNIST digit recognition: The MNIST is a standard benchmark for classification problems that categorizes handwritten digits, each of size 28×28 [69]. A simple 1-layer NN with 784 inputs and 10 outputs was trained - first without and then with the 3 types of regularization.

Table 3.3 summarizes the benefits of approximating the weights of the NN for all types of penalty functions and select values of error threshold ϕ. The first column shows the initial energy levels before optimization (but with BMS in place). The ones with regularization are lesser than those without as the moving away of

weights from 0 decreases the average energy of the BMS. This is evident from the nature of the plot in fig. 3.3(a). It must be mentioned that the classification error rate does not change with just the incorporation of BMS, as weight values remain exactly the same. Significant energy savings were obtained even for $\phi = 0$ owing to certain degree of redundancy in some inputs. The entire data has been plotted in fig. 3.7. All accuracy and energy consumption values (here, and henceforth) are for 64-bit long SNs; the latter goes up linearly with the length. The BMS which represented the 784 inputs had a constant share of 30.5 nJ (averaged over all test samples) for all types of regularization and all values of ϕ, and the rest of the energy was from those of the 748×10 weights.

			AV			CS	
Reg.	ϕ	–	0	0.05	0.10	0	0.02
None	Energy	355.8	282.5	226.1	200.9	204.2	191.2
	Error	11.98	12.11	15.15	18.66	19.10	20.15
Type 1	Energy	349.4	276.8	225.6	202.9	197.4	187.4
	Error	12.83	13.07	16.98	21.44	21.99	22.59
Type 2	Energy	342.9	272.8	216.0	192.0	192.0	181.7
	Error	12.75	12.92	16.42	22.01	20.38	20.95
Type 3	Energy	349.8	276.4	215.0	187.8	193.6	180.7
	Error	12.35	12.64	16.52	20.87	19.29	19.92

Table 3.3: Variation of 1-layer network energy (in nJ) and classification error rate (in %) on the MNIST test dataset with different values of error threshold ϕ with both AV and CS types of constraint. The 1^{st} column of data corresponds to BMS without any weight approximation.

As can be seen, when AV constraint is employed, the trade-off with type 3 regularization is comparable (or slightly better) to that without for somewhat large values of ϕ. However, with CS, type 3 is markedly better than others, and also beats the AV. It must however be noted that the CS constraint brings about a sharp reduction in energy accompanied by a significant increase in classification inaccuracy with the smallest value of error threshold ($\phi = 0$). On the other hand,

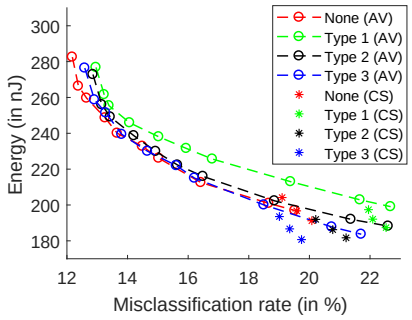


Figure 3.7: Energy vs classification error rate curve for the MNIST dataset 1-layer NN. The dots are for the AV constraint, whereas the asterisks are for CS. Optimization was done with 1000 training samples.

AV provides a more gradual trade-off with more control on the misclassification rate. This is due to the former bounding network outputs from only 1 side, leading to a larger solution space.

The latency of inference is $8.21ns$ per bit of the SN (as in sec. 3.3.3); thus a bitstream length of 64 would imply that the classification of 1 sample requires $0.525\mu s$.

For the 2-layer NN, input images were scaled down to size 14×14 to reduce the time required to solve the problem, and 25 neurons were used in the hidden layer. Characteristics of the network before and after weight optimization, are summarized in Table 3.4. In all energy values, input BMS consumed 8.25 nJ; remaining

			AV			CS	
Reg.	ϕ_W	-	0	0.05	0.10	0.00	0.01
None	Energy	221.2	191.3	159.6	147.7	188.1	172.5
	Error	6.97	6.79	7.08	7.32	8.18	8.13
Type 1	Energy	216.1	187.1	156.0	146.6	184.4	168.3
	Error	7.43	7.45	7.83	7.85	8.75	8.86
Type 2	Energy	210.2	181.8	148.4	138.8	179.7	162.3
	Error	7.09	7.08	7.21	7.41	8.02	8.17
Type 3	Energy	212.7	188.4	156.4	146.2	185.5	171.0
	Error	7.13	7.13	7.35	7.45	7.99	8.05

Table 3.4: Results for the MNIST 2-layer network for select values of error threshold of the outer layer (ϕ_W). Classification error and energy (in nJ) are for 64-bit long SNs.

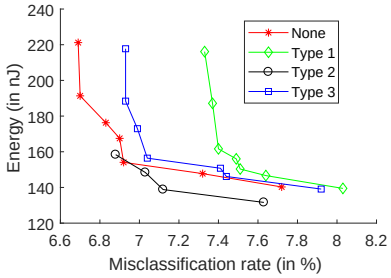


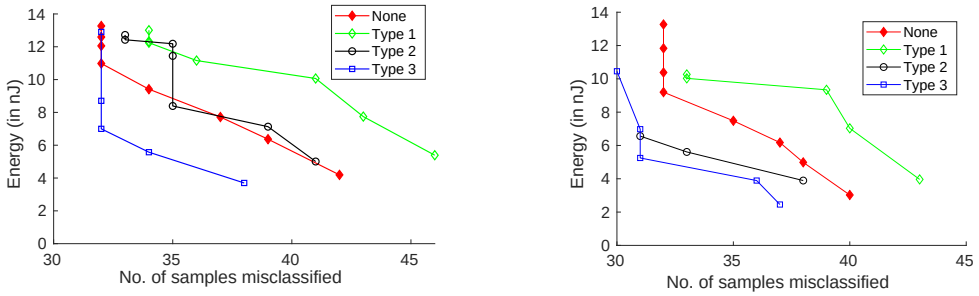
Figure 3.8: Plot of classification error rate against energy for 2-layer NN of MNIST dataset with AV constraint.

was distributed between hidden and output layer BMS roughly in the ratio 18 : 1. The misclassification error rates with floating point double precision and without any weight approximation are 6.69% without regularization, and 7.33%, 6.98%, and 6.93% for types 1,2 and 3 respectively. These are reasonably close to the corresponding values with 64-bit long SNs (1^{st} column of table 3.4).

The energy-error trade-off with the Absolute Value constraint is depicted in fig. 3.8. For each kind of regularization (including none), only the points which are pareto-optimal have been jotted (that is to say, energy-error pairs having higher values of both than at least 1 other pair have been skipped). Type 1 and type 3 of regularization do not exhibit lesser values of both energy and error than the case with None. However, type 2 possesses similar or more optimal values for somewhat high values of ϕ_W. A reduction of 40.5% in energy is observed with $\phi_W = 0.15$ for a degradation of about 1% in accuracy. With the CS constraint, although energy values show a significant dip from those prior to optimization, the error that creeps in is much higher than that with AV. The distribution of weights in both layers of this NN (as well as in the NNs of the next datasets) was similar to those of the 1-layer counterpart.

2. Wine Quality: This dataset (as well as the next one) was obtained form

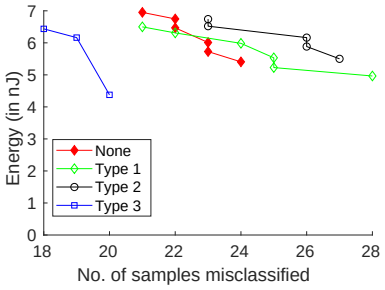
the UCI Machine Learning Repository [78]. The goal here is to train a network to estimate the quality of samples of red wine on the basis of results of physiochemical tests [29]. Only a 2-layer NN with 12 input parameters and 20 hidden neurons was trained with 1249 samples and tested on 250 samples. The no. of misclassified samples before weight approximation and using floating-point precision was 31, 34, 32, and 32 without and with the 3 types of regularization respectively. Fig. 3.9 plots the energy-error curve for both constraints. As is evident, the type 3 penalty function provides more optimal pairs of energy and error values than the others in both cases; with the CS constraint surpassing the AV.



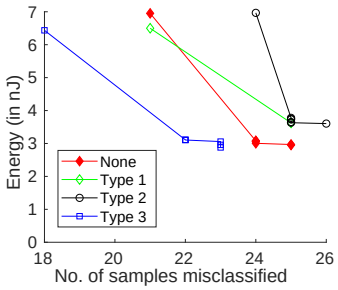
(a) Absolute Value constraint **(b)** Class. Specific constraint

Figure 3.9: Trade-off between network Energy and classification error rate for the Wine Quality test dataset. For each curve in (a), the topmost point (one with the highest energy) corresponds to the values before optimization. 0.46 nJ of energy was for the inputs; hidden and output layer weights' consumption ratio was roughly 2:1.

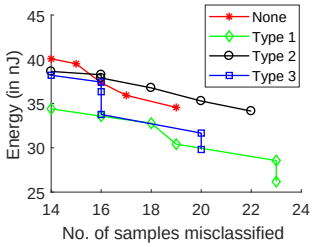
3. SONAR, Rocks vs Mines: This is about distinguishing between metal surfaces and rocks using sonar signals bounced off from them [41]. Both the training and test datasets contain 104 samples, each having 60 inputs. Both a 1-layer NN and a 2-layer NN (with 15 hidden units) were trained. The results are plotted in fig. 3.10. Before weight approximation, the no. of misclassified samples, with floating point double precision, were 20, 21, 24, & 18 for the 1-layer NN and 15,14,14, & 13 for the 2-layer NN for None, type 1,2 and 3 respectively.



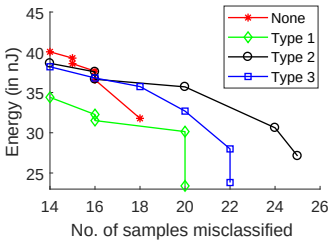
(a) Single-layer NN - AV Constraint



(b) Single-layer NN - CS constraint



(c) Two-layer NN - AV Constraint



(d) Two-layer NN - CS constraint

Figure 3.10: Energy v/s inaccuracy in classification for the SONAR dataset. (a)-(b) 1-layer NN, and (c)-(d) 2-layer NN. Input BMS required 2.29 nJ. Hidden and output weights in 2-layer NN used energy in ratio about 35 : 1.

For the 1-layer NN, with both the AV and CS constraint (figs. 3.10(a) and (b)), type 3 works the best whereas types 1 and 2 are either similar or worse than None. The CS constraint is better than the AV for all types except type 3, where they are comparable.

In the 2-layer NN, types 1 and 3 outperform None when AV constraint is used (fig. 3.10(c)), whereas type 2 is a bit worse. With the CS constraint (fig. 3.10(d)), only type 1 appears to be better than None. Among the constraints, the latter provides better trade-off with no regularization and type 1; but the two are comparable when types 2 and 3 are used.

3.7 Conclusion

This chapter proposes the use of Magnetic Tunnel Junctions as Stochastic Number Generators in an SC based hardware implementation of Neural Networks. We design an energy-efficient version of an MTJ-SNG (named BMS) that significantly reduces the average energy per bit of a stochastic stream and propose its use in an SC-based NN. We go on to develop an algorithm based on convex optimization that aims to adjust the weights in such an NN in a way that brings about a reduction in the energy consumption. This approximation leverages the error resilient nature of applications of NNs. The algorithm would be applicable to not only feed-forward networks, but also other more complicated architectures (such as Convolutional and Recurrent NNs) since the basis of achieving energy efficiency remains the same.

Further, we propose 3 types of penalty functions to be used for weight regularization during training of the NNs, keeping in mind the kind of weight distribution that leads to lower energy. Lastly, we suggest a small modification to constraints in the optimization procedure that caters to classification-based problems by taking advantage of a certain redundancy in their outputs. To give a perspective of the benefits brought about by our approach, the proposed algorithm brings about a 40% reduction in energy consumption with less than 1% accuracy loss on the 2-layer MNIST network. Future work could propose other optimization methods that can better workaround the non-convexity of NNs and approach the problem in a wholesome way. Also, more efficient ways of using the MTJ as an SNG could be developed.

Chapter 4: In-situ Training of MTJ Neural Network Crossbar

In the previous chapter, we proposed the energy-efficient use of MTJs as Stochastic Number Generators in an NN architecture which operates in the Stochastic Computing (SC) domain. While SC does involve smaller designs with low power consumption, the requirement of long bitstreams for good accuracy is a hindrance to fast execution times [12]. In this chapter, we consider the resistive crossbar architecture described in sec. 2.3.1 for the physical realization of NN weight matrices due to its inherent parallelism. We investigate into MTJ crossbars and state the drawbacks of training them as one would normally do (that is, doing the training in software and then programming the crossbar). We then propose techniques for on-chip training of the crossbar that takes care of those issues.

4.1 Introduction

The emergence of novel devices and special-purpose architectures has called for a shift from conventional digital hardware for implementing neural algorithms [121]. Attempts have been made towards dedicated hardware designs and realization of the synaptic weights (and neurons) of a Neural Network (NN) by using CMOS transistors in an analog fashion [87, 97]; but these have met with challenges of scalability and volatility. Parallel research work has focused on using post-CMOS

devices such as memristors, which are non-volatile devices with a variable resistance, as synaptic weights [102]. In an analog computing framework, the conductance of resistive devices encodes the NN weight value. However, the fabrication of multilevel memristors with stable states is still a challenge [142, 96]. Another choice is the Magnetic Tunnel Junction (MTJ) [132]. Its non-volatility and scalability make it a particularly lucrative choice for in-memory processing type of architectures for neural networks.

Neural network architectures can be realized with resistive devices using the crossbar configuration which allows greater scalability and higher performance due to its inherent parallelism [102, 141, 130, 113, 86]. The crossbar not only stores the weight values but also does the matrix-vector computation of the output in-memory. This obviates the need for fetching the weight values from the memory into the processing unit.

The existence of only 2 stable states in MTJs makes them a good candidate for the realization of binary weight networks. Obtaining optimal weights for a binary network in software can be impractical because its discrete nature requires integer programming. One way of training such NNs is to perform weight updates stochastically, which is justifiable from evidences that learning in human brains also has some stochasticity associated [121]. That such a method can lead to convergence with high probability in a finite time has been shown in [112], although using the perceptron learning rule. Also, when physically realizing an NN on hardware, the underlying device variations can have a substantial impact on the model accuracy, and need to be accounted for in the training process. Merely characterizing the variations in the hardware platform is not sufficient for overcoming this issue.

Efficient architectures for the realization of different types of network models, such as convolution and recurrent neural networks [139], Liquid State Machines [57] and Echo State Networks [48], are also being investigated. Neftci et. al. [94] construct a Restricted Boltzmann Machine (RBM) with Integrate & Fire Neurons and present an event-driven variation of the Contrastive Divergence (CD) learning algorithm. Herein the recurrent structure of the network is exploited to mimic the construction and reconstruction phases of CD weight update in a spike-driven fashion, and STDP is used to carry out the weight updates. In [116], an approach to implement CD in one layer of an RBM with memristors as synapses is presented. However, the RBM has stochastic binary units and weight updates are ternary. Suri et. al. [120] fabricate an HfO_x device and test it for synapse implementation, internal neuron-state storage and stochastic neuron activation function of a hybrid RRAM-CMOS RBM architecture.

In this chapter, we explore the use of MTJ crossbars for the hardware implementation of the synaptic weight matrices of feed-forward neural networks and RBMs. Our contributions are as follows:

- We propose the on-chip or *in-situ* training of these MTJ crossbars, which allows us to exploit their inherent parallelism for significantly faster training and also accounts for device variations.

- We advocate a probabilistic way of updating the MTJ synaptic weights of an NN through the gradient descent algorithm by exploiting the stochasticity in their switching.

- We experiment with two crossbar structures: with and without access transis-

tors. The latter poses the additional challenge of sneak-path currents during programming which makes training in-situ the only choice to achieve satisfactory performance.

- Then we go on to propose a modification of the Contrastive Divergence algorithm that is to be adopted when the MTJ crossbar is used to implement an RBM, and a means of using MTJs for storing RBM hidden units' states.

- Finally, we support our proposed techniques with data by modeling device and circuit properties and running simulations.

4.2 Background

4.2.1 Crossbar Architecture for Neural Networks

The basic structure of a feed-forward neural network and operations for inference and training were discussed in sec. 2.1. Recall that both forward and backward propagation require a matrix-vector multiplication for each layer in the NN (eqns. 2.2 and 2.4).

The computational complexity is therefore $O(M.N)$, for a layer with M inputs and N outputs, for an implementation on general-purpose hardware. The crossbar architecture discussed in sec. 2.3.1 is well-suited to perform mat-vec multiplications in the analog domain since it offers a high level of parallelism. It has repeatedly been proposed as an accelerator for NN implementations since the inference operation in a layer can be done in $O(1)$ time.

It is worth noting though from eqn. 2.3 that the weight update of a synapse

is local in nature, in the sense that it depends only on the information available at that synapse - the input to it and the error at its output. This motivates the development of techniques for performing even the weight updates (and not just inference) in parallel.

4.2.2 Related Work

Several studies have investigated how a crossbar array with memristors [103, 108, 17, 133], MTJs [142, 130] and domain-wall ferromagnets [110, 108] can implement Spiking Neural Networks (SNN) trained using Spike-Timing Dependent Plasticity (STDP). Srinivasan et. al. [119] propose the use of a pair of MTJs for a synapse in an SNN - one for long-term and the other for short-term synaptic memory. The literature also contains several works [38, 58, 138] considering supervised learning of SNNs for various reasons.

Many works have dealt with methods and algorithms for training networks by modifying their weights at the site of their occurrence [22], instead of doing it offline. Hasan et al. [47] and Soudry et al. [118] have implemented multi-layer NNs on memristive crossbars trained on-chip using the backpropagation algorithm and demonstrated on supervised learning tasks. Gokmen et. al. [40] use stochastic computing techniques for parallel weight update on crossbar arrays - numbers that are encoded from neurons are translated to stochastic bit streams, with device conductance changing when the streams coincide. In [72], hybrid semiconductor/nanodevice technology neural nets with binary synapses were trained "in-situ" using the error backpropagation rule, and the results obtained were almost at par with networks with continuous weights trained in software.

Continuous weight networks can be simplified into discrete weight networks without significant degradation in classification accuracy while achieving substantial power benefits [105]. The use of discrete weight networks, such as BinaryConnect [30] and in [76], also stems from the challenge to address the high storage and computational demands of a large number of full-precision weights. Ni et. al. [96] design a distributed in-memory computing architecture based on binary RRAM-crossbars for memory and logic units.

4.3 MTJ Crossbar based Neural Networks

The stochastic switching nature of MTJs has necessitated the usage of high write currents or write duration in memory applications to ensure low write errors. Alternatively, one can also use them to implement the synaptic weights in a crossbar where each cross-point would be an MTJ in one of its 2 states. They are capable of being programmed with high speeds and exhibit endurance of the order of 10^{15} write cycles. However, the inherently binary nature of MTJs implies that such synapses can represent only 2 weight values and hence can implement only binary networks. Although it is possible to have some continuous behavior with the inclusion of a domain wall in the free layer [110], the maturity of such technology is not at par with that of the binary version [108].

4.3.1 Training Binary Networks

Obtaining optimal binary weights for an NN is an NP-hard problem with an exponential time complexity [112], and hence a solution must involve training of the

binary network of some form. This prompts the use of a probabilistic learning technique since the required weight update is continuous whereas any possible change in the conductance of the MTJ could only be discrete, in fact binary. As stated in [121], stochastic update of binary weights is computationally equivalent to deterministic update of multi-level weights at the system level.

In [130], Vincent et al. exploit the stochastic switching behavior of MTJs to propose its use as a 'stochastic memristive synapse' in an SNN taught using a simplified STDP rule. However, there is no theoretical guarantee of the convergence of STDP for general inputs [74], and Lim et. al. [79] believe that the learning performance using STDP in still in its early stages. We propose using a probabilistic learning approach by training using the gradient descent method (which requires weight updates of the form in eqn. (2.3)) as demonstrated in section 4.4.2.

4.3.2 The Motivation for In-situ Training

There are 2 ways (primarily) in which MTJs in the crossbar can be connected to their respective input and output terminals -

1. With selector devices (1T1R) - Here each MTJ synapse is connected in series with an MOS transistor (as in fig. 4.1), resulting in $O(M \times N)$ transistors in the crossbars.
2. Without selector devices (1R) - Synapses are directly connected to the crossbar terminals; there are no transistors within the crossbar, such as the one in fig. 2.5. While a 1R structure provides greater scalability, it does so at the cost of reduced control of and access to individual synapses.

Stochastic learning can be done (simulated) offline and the final weights ob-

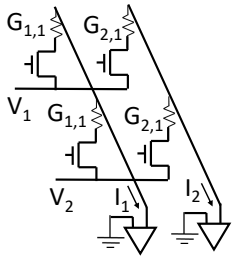


Figure 4.1: A 2×2 crossbar with selection transistors

tained can be programmed on to the crossbar deterministically. But, since MTJs have an inherently stochastic switching behaviour, deterministically programming them on a crossbar would require currents having high magnitude and duration to guarantee successful write operations. The possibility of selecting synapses to be written in the 1T1R architecture ensures no *side-effects* of this method stemming from alternate current paths (because there would be none). But, despite circumventing this issue, this architecture can suffer from performance degradation due to the intrinsic device variations which only aggravate with scaling. On the other hand, in a 1R architecture, such high programming currents, when they sneak through alternate paths, are bound to cause unwanted changes in neighboring synapses owing to which the weights may never converge. This necessitates *in-situ* training of the crossbar in a probabilistic way for both 1T1R and 1R configurations, as only training on the hardware can account for both alternate paths and device variability.

4.3.3 Network Binarization and MTJ as a synapse

Simply using ± 1 as the binary weight values, represented by the P and AP states of an MTJ, is naive and estimating a good scaling factor b is essential for overall network performance. An appropriate way to determine a suitable b is to minimize the L2 loss between the real-valued weights W and quantized ones, as was done in

[105]. This provides a solution $b = \|W\|_1/n$, which is the mean of absolute values of W (n being the no. of elements in W). Thus an MTJ in the P (AP) state would signify a weight of $+b$ ($-b$).

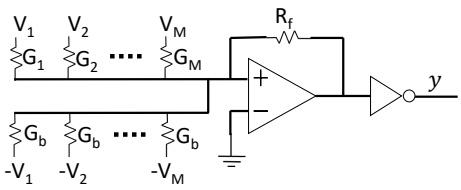
The weights of an NN are almost always bipolar whereas the conductance of an MTJ or memristor is always positive. One method to realize negative weights is to effectively offset the conductance with a fixed bias; in the case of MTJs, we choose $G_{bias} = (G_P + G_{AP})/2$ as it brings symmetry to the effective conductance: $G = G_P - G_{bias}$ would correspond to the positive weight, say $+b$, and $G = G_{AP} - G_{bias}$ would correspond to the negative weight, say $-b$. These bias resistors are fed with the negative of the input voltages, and the output current in any column of the crossbar can be obtained by adding the bias currents to the current received from the MTJ synapses. That is, the total output current can be written as

$$I = \sum_i (G_i - G_{bias})V_i = \sum_i (G_i V_i + (-G_{bias} V_i)) = \sum_i (I_i + I_{bias,i}) \tag{4.1}$$

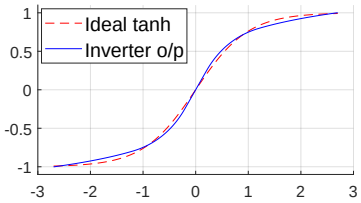
Fig. 4.2(a) depicts the implementation of eqn. (4.1), with the inverter producing the response of the *tanh* activation function (which we have used in all our NNs) as shown in fig. 4.2(b). The average and maximum errors between the ideal value and the inverter output are 0.0327 and 0.0606 respectively.

4.4 In-situ Training of the NN Crossbar

We first provide a high-level understanding of how an MTJ synaptic crossbar implementing a feed-forward NN should work. For the sake of simplicity, all operations are described for a single-layer NN and can be easily scaled to multiple layers (more



(a) An M-input neuron with MTJ synapses and bias resistors, with an inverter for the activation function



(b) Transfer characteristics of the inverter

Figure 4.2: (a) Synaptic weights and activation function in each column of the crossbar. G_b is the constant resistor which creates the bias. (b) Comparison of the output characteristics of the inverter and the actual *tanh* function. Producing this behavior requires the relation $MR_fV_{rd}(G_P - G_{AP}) = 7.2$ to be satisfied with inverter output load of $10k\Omega$. V_{DD} and V_{SS} of inverter are $1.8V$ and $-1.8V$ respectively.

details subsequently). We then illustrate how the gradient descent method can be used for the stochastic weight update of MTJs, and finally describe the in-situ training procedure for the 2 crossbar architectures.

4.4.1 Overview of Operations

The training process is carried out as follows.

Read Phase: Upon receiving a training input $x \in R^M$, the input terminals are applied with voltages $V_i^r \in [-V_{rd}, V_{rd}] \ \forall \ i$ proportional to x_i, whereas the output terminals are maintained at ground potential. Current $I_{ji} = G_{ji}V_i^r$ flows through the (j, i) synapse and the total current I at the output terminals are suitably converted to output y.

Write Phase: Using y and the desired output, calculate the error δ. Table

Input	Error	ΔW	W and G	Switch
$x > 0$	$\delta > 0$	$\Delta W < 0$	*Decreases*	$P \rightarrow AP$
$x > 0$	$\delta < 0$	$\Delta W > 0$	*Increases*	$AP \rightarrow P$
$x < 0$	$\delta > 0$	$\Delta W > 0$	*Increases*	$AP \rightarrow P$
$x < 0$	$\delta < 0$	$\Delta W < 0$	*Decreases*	$P \rightarrow AP$

Table 4.1: The write phase. Signs of x, δ, and ΔW, required change in weight W and conductance G, and the desired direction of switching of MTJ Synapse

4.1 lists the 4 possible cases of weight update depending on x and δ. The gradient descent algorithm requires a weight update of the form of eqn. (2.3). An appropriate way to realize this, as suggested in [73], is to set switching probabilities proportional to (the magnitude of) Δw calculated in eqn. (2.3). Our way of achieving this is explained next.

The process of read and write are carried out for each input sample and repeated for several iterations until convergence is achieved.

4.4.2 Stochastic Learning of an MTJ Synapse

We will now describe how the stochasticity of MTJ switching can be used to perform weight updates with gradient descent method. Just as the weight update in eqn (2.3) is a function of 2 variables (the input and the error), the probabilistic switching of MTJs can be controlled by 2 physical quantities- the magnitude and the duration of the programming current. We choose the magnitude of the write current to be dependent on the input x_i and the duration on the error δ_j. However, as can be evidenced from eqn (2.5) and fig 2.4, the switching probability P is a highly non-linear function of the parameters a and t (recall $a = I/I_{c0}$), whereas the desired probability, being proportional to ΔW_{ji}, is a linear function of x_i and δ_j. Further, the switching probability does not immediately rise with the pulse width and the write current as they increase from 0, indicating some kind of soft threshold. Note that the direction of switching can be decided by the polarity of the write current.

We therefore model switching probabilities by a linear mapping of x and δ to write current I_{wr} and duration t_{wr} respectively as follows. Usually $|x| \leq 1$, and henceforth assume for simplicity that $|\delta| \leq 1$ (can be ensured by normalizing and

adjusting with η). The pulse width t_{wr} is set at a minimum of t_0 and increases linearly with $|\delta|$ (since t_{wr} needs to increase irrespective of the sign of δ) as

$$t_{wr} = t_0 + t_1|\delta| \tag{4.2}$$

Similarly, the write current (I_{wr}) would be a minimum of I_0 and increase linearly with $|x|$ as

$$I_{wr} = I_0 + I_1|x| \tag{4.3}$$

We now wish to find coefficients t_0, t_1, I_0 and I_1 that yield MTJ switching probabilities (P) close to the desired probabilities of weight update. A certain probability of switching can be obtained for different combinations of I and t, as is evident from fig. 2.4. We first fix the range of pulse widths by choosing suitable t_0 and t_1 (refer to table 4.3). We want a nearly 0 switching probability for $t_{wr} = t_0$ irrespective of the value of I_{wr} because $\Delta W = 0$ for $\delta = 0$ regardless of x. We thus choose the maximum I_{wr} (which is $I_0 + I_1$) to be that value of I for which the plot of P against t_{wr} starts rising at t_0. That is

$$P(I_0 + I_1, t_{wr}) \quad is \begin{cases} < P_0 & \text{for } t_{wr} < t_0, \\ \geq P_0 & \text{for } t_{wr} \geq t_0 \end{cases} \tag{4.4}$$

where P_0 is a small value. So now even if $|x|$ is (as high as) 1, $P = P_0$. In our experiments, we chose P_0 to be about 0.05.

A symmetric argument holds when $x = 0$. For $t_{wr} = t_0 + t_1$, we want $P \approx 0$ if $I_{wr} = I_0$, (because $\Delta W = 0$ for $x = 0$). But P should start increasing as soon as

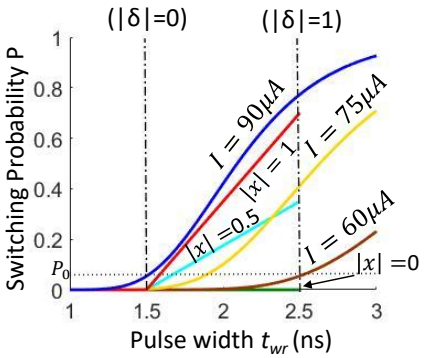


Figure 4.3: P vs t_{wr} of the linear model and desired probabilities (obtained with $\eta = 0.7$) for $AP \rightarrow P$ transition. The region between the dashed vertical lines is of interest. The dark green, cyan and red straight lines plot desired probabilities for $|x| = 0, 0.5$, and 1 respectively. The brown, yellow and blue plots correspond to the actual switching probabilities (obtained from the linear model) for the mapped currents $I = 60\mu A, 75\mu A$, and $90\mu A$

Weight Update	MTJ Switching
$\|\delta\| = 0$	$t_{wr} = t_0$
$\|\delta\| = 1$	$t_{wr} = t_0 + t_1$
$\|x\| = 0$	$I_{wr} = I_0$
$\|x\| = 1$	$I_{wr} = I_0 + I_1$

Table 4.2: Boundary values of the parameters in the weight update eqn. (2.3) and their counterpart in probabilistic switching of MTJ.

Direction	$AP \rightarrow P$	$P \rightarrow AP$
t_0	$1.5ns$	$1.5ns$
t_1	$1ns$	$1ns$
I_0	$60\mu A$	$140\mu A$
I_1	$30\mu A$	$60\mu A$

Table 4.3: The coefficients that fit the model for both $AP \rightarrow P$ and $P \rightarrow AP$ switching

I_{wr} increases, that is

$$P(I_{wr}, t_0 + t_1) \quad is \quad \begin{cases} < P_0 & \text{for } I_{wr} < I_0 \\ \geq P_0 & \text{for } I_{wr} \geq I_0 \end{cases} \tag{4.5}$$

Fig 4.3 shows how well the linear model approximates the required $AP \rightarrow P$ switching probabilities (similar curve fitting for $P \rightarrow AP$ as well). Table 4.2 shows the write currents and duration for boundary values of $|x|$ and $|\delta|$ and table 4.3 lists the values of the coefficients in eqns. (4.2) and (4.3). One could use non-linear models for mapping $|\delta|$ and $|x|$ to t_{wr} and I_{wr}, respectively, in order to better fit the desired switching probabilities; however, that would complicate the analog circuit responsible for the conversion. Owing to this, and the closeness with which the

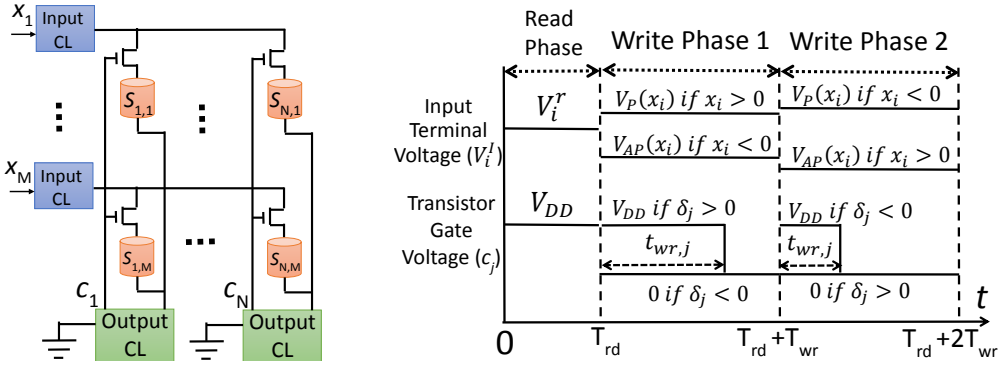
linear model can replicate the stochastic switching characteristics, we stick to the linear version.

While training neural nets, it is necessary to have a small learning rate to avoid getting stuck in local minima. In our training strategy, this means having small probability values of MTJ switching. On the other hand, it is necessary to ensure that the probabilities are not so low that they barely cause any changes in the weight values. As can be seen from fig. 4.3, in the average case of having $|x| = |\delta| = 0.5$, the $AP \rightarrow P$ switching current and pulse width are $I = 75\mu A$ and $t_{wr} = 2.0ns$ respectively and the switching probability stands at around 10%. The model parameters t_0, t_1, I_0 and I_1 have been adjusted so that probability values are within a reasonable range, that is, neither too small nor too large, and help the training process to converge.

Next, we describe the 1T1R and 1R crossbar architectures implementing the NN. We show how these can be trained in-situ using the stochastic learning technique described above.

4.4.3 The 1T1R Architecture

This is the conventional architecture for memory applications where each cell has a selection transistor. One major advantage of being able to selectively turn off certain cells is that it disallows the presence of undesired sneak currents which lead to unnecessary power consumption at a minimum. Fig 4.4(a) shows a 1T1R crossbar where each MTJ synapse is connected in series with an NMOS transistor. Input and output terminals are interfaced with necessary Control Logic (CL). All the transistors in a single column will have a common gate voltage since the corresponding



(a) An $M \times N$ crossbar **(b)** Write voltages and control signals.

Figure 4.4: The 1T1R crossbar. (a) Schematic (b) Read & write phases signals

synapses are connected to the same neuron output, and hence will always have the same error 'δ' and write pulse width t_{wr}.

Fig 4.4(b) plots the signals during both the read and write phases. During the read phase ($0 \leq t \leq T_{rd}$), all transistors are turned on: $c_j = V_{DD} \quad \forall \quad j = 1...N$ so that all columns (neuron outputs) are read simultaneously. Inputs x_i are provided to their respective input CLs which convert them to read voltages V_i^r. Output currents I_j are processed by the output CLs.

4.4.3.1 Updating the crossbar

Decide the write currents that should be provided to each input row and the pulse widths for each output column as described in sec. 4.4.2. Recall that the former depend on x and the latter on δ. The direction of the currents would depend on the sign of the desired weight update. Apply suitable write voltages at the input terminals while grounding the output terminals to 0.

For the (j, i) synapse, the write pulse width depends on only $|\delta_j|$, and the write current magnitude depends on $|x_i|$. But the direction of switching depends on the signs of δ_j and x_i (see Table 4.1) and has to be decided by the polarity of current.

For eg. two MTJ synapses belonging to the same row but different columns may have opposite signs of δ. Thus, despite having the same input x_i, they are required to switch in opposite directions and hence need write voltages of opposite sign. This requires us to split the write phase into two parts as explained next.

Since the transistor gate control signals are connected to the output CLs, we can select or deselect a certain column based on information at its respective CL, which is the error δ. We therefore program the crossbar sequentially in 2 stages, with the columns updated in a given stage depending on the signs of δ. Each phase has a duration of T_{wr} (which need not be more than $t_0 + t_1$, see eqn. (4.2)). The voltage signals in each phase are plotted in fig. 4.4(b) and detailed below -

1. Phase 1: $T_{rd} \leq t \leq T_{rd} + T_{wr}$. Update the weights of the columns which had $\delta > 0$. Then, the transistor control signals would be

$$c_j = \begin{cases} V_{DD}, & \text{for } \delta_j > 0 \textit{ and } 0 \leq t - T_{rd} \leq t_{wr,j} \\ 0, & \text{for } \delta_j < 0 \textit{ or } t_{wr,j} \leq t - T_{rd} \leq T_{wr} \end{cases} \tag{4.6}$$

   And the write voltages applied at the input terminals would be

$$V_{wr,i} = V_P(x_i)u(x_i) + V_{AP}(x_i)u(-x_i) \tag{4.7}$$

   where u is the unit step function.

2. Phase 2: $T_{rd} + T_{wr} \leq t \leq T_{rd} + 2T_{wr}$. Update the weights of those columns which had $\delta < 0$. Here, the signals are opposite to those in phase 1 as shown in fig. 4.4(b).

Here V_P (V_{AP}) is the voltage applied to switch from P$\rightarrow$AP (AP$\rightarrow$P) and can be

obtained using eqn. (4.3) and $R_P(R_{AP})$. V_P and V_{AP} still depend on $|x_i|$, but for brevity explicit mention will be omitted henceforth. Let MTJs in the crossbar be arranged in a way that positive (negative) current from the i^{th} input terminal to j^{th} output terminal can switch $S_{j,i}$ from $P \rightarrow AP$ ($AP \rightarrow P$); hence $V_P > 0$, ($V_{AP} < 0$). Parameters in table 4.3 give $V_P \in [0.68, 0.98]$ volts and $V_{AP} \in [-0.81, -0.62]$ volts.

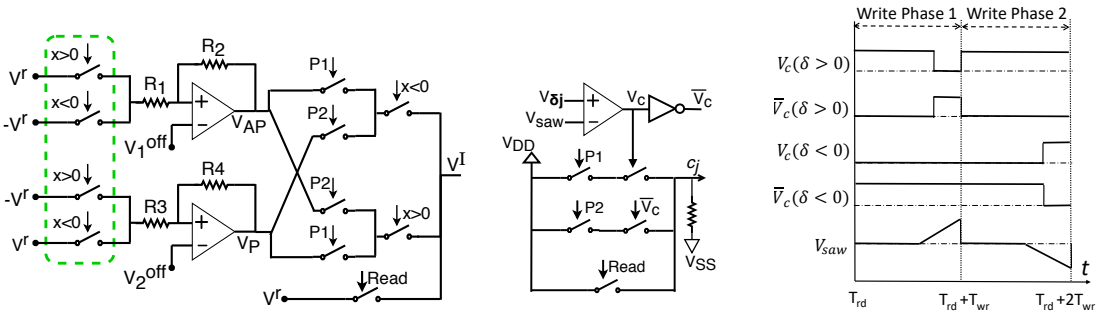
Thus we can see that the read and update operations are completed in $T_{rd} + 2T_{wr}$ time which is $O(1)$. The weight update is sequential with respect to the sign of δ, but it is done in parallel for all those columns that have the same sign of δ.

4.4.3.2 Control circuits

Fig. 4.5 shows the internals of the Input and Output CLs. In fig. 4.5(a), in the Read Phase, the read voltage V^r is directly passed on. The write voltages V_P and V_{AP} are obtained by suitably scaling V^r or $-V^r$, and shifting that by an offset to reach the desired range of values. Due to opposite polarities, V_{AP} is always obtained from a positive V^r, and V_P from a negative V^r, with the switches in the dashed green box thrown as per the sign of x. Switches controlled by P1 and P2 are 'on' in Write Phases 1 and 2 respectively, and 'off' otherwise. In our design, $V^r \in [-V_{rd}, V_{rd}]$ with $V_{rd} = 0.2V, R_2/R_1 = 0.95, R_4/R_3 = 1.5, V_1^{off} = -0.318V, V_2^{off} = 0.272V$.

Fig. 4.5(b) depicts the control logic in the output terminals to decide the duration of Read and Write phases by controlling the crossbar transistors' gate voltage c_j. We have $V_\delta \propto \delta$ (through circuits described in sec. 4.4.5). If $\delta > 0$, V_C is high for some part of write phase 1 (as per eqn. (4.6)) which pulls c_j up to V_{DD} for the same duration. Similarly, for $\delta < 0$, $\overline{V}_C$ and c_j are high for a part of write phase 2. Fig. 4.5(c) illustrates the timing diagram of V_C and its complement for the

2 possibilities of δ, and also of the sawtooth waveform for generating V_C. In Phase 1, V_{saw} stays 0 until time $t_0 = 1.5ns$ and then rises linearly to the maximum value of V_δ. In Phase 2, the behavior is same but with an opposite polarity.



(a) Control Logic in the crossbar input terminals. **(b)** Output CL **(c)** Write control signals

Figure 4.5: Circuit of the crossbar's (a) Input, and (b) Output CLs, and (c) Write phase signals timing diagram

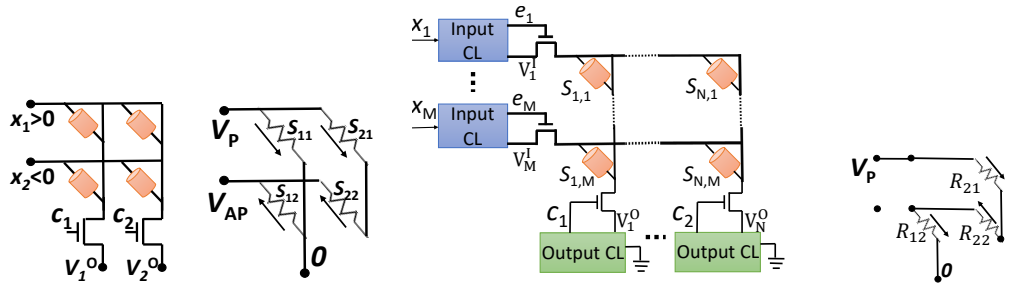
Due to limitations on the scalability of 1T1R architecture, it is worth exploring the feasibility of transistor-less crossbars to achieve even higher density of integration.

4.4.4 The 1R Architecture

Eliminating the need to have an access transistor for every synapse in the crossbar will allow for compact designs having an integration density of about $4F^2$/device. But the inability to select the synapses to be updated during programming results in leakage currents through alternate paths that not only waste energy but also can lead to undesirable changes in synaptic conductance. We first see the effect of such currents with the previously proposed write-strategy and then suggest a modified strategy (and circuit) for the 1R architecture.

4.4.4.1 Two-phase update

Let's analyze the impact of sneak paths on the 1R crossbar with the 2-phase update strategy used previously. We first demonstrate the presence of sneak paths with a small example. Fig 4.6(a) shows a 2×2 crossbar with transistors only at the output terminals (to choose columns to be written in any particular phase). Assume without loss of generality that a certain input x with $x_1 > 0, x_2 < 0$ produced errors $\delta_1 > 0, \delta_2 < 0$ at the outputs. The equivalent circuit during write phase 1 is drawn in fig. 4.6(b). It depicts the currents through the synapses, with the ones through S_{21} and S_{22} being undesired. These *may* falsely switch S_{21} from $P \rightarrow AP$ and S_{22} from $AP \rightarrow P$ if they are in P and AP states respectively.



(a) **(b)** **(c)** An $M \times N$ crossbar with 1R structure **(d)**

Figure 4.6: (a) and (b) Alternate current paths in the 1R structure with 2-phase write strategy - (a) A 2×2 crossbar. (b) Its equivalent circuit in write phase 1 with $c_1 = V_{DD}, c_2 = 0, V_1^O = 0, V_1^I = V_P, V_2^I = V_{AP}$. (MTJ synapses shown as resistors). (c) Schematic of the proposed 1R Architecture for MTJ crossbar, (d) The equivalent circuit in phase 1 with 4-phase writing.

We now state a worst-case scenario for a crossbar with M inputs. If M is large, analysis using Kirchhoff's Current Law shows that the potential difference across an MTJ synapse could go as high as $(V_P - V_{AP})$. The current through such an MTJ, if in the P state, is $I = (V_P - V_{AP})/R_P$ and is high enough (recall $V_{AP} < 0$) to switch it from $P \rightarrow AP$. In the other extreme case, a potential difference of $(V_{AP} - V_P)$

leading to current $I = (V_{AP} - V_P)/R_{AP}$ through an MTJ in the AP state will switch it from $AP \rightarrow P$.

It is also necessary to mention an average (expected) case. Here these currents reduce to $I = (V_P - V_{AP})/2R_P$ and $I = (V_{AP} - V_P)/2R_{AP}$, respectively, which are half of those found previously, but still have some probability of switching MTJs (because these currents are roughly the same as V_P/R_P and V_{AP}/R_{AP}). Thus, chances of unwanted flips of MTJs are quite significant, which calls for some modification in the circuit and/or in the programming method.

4.4.4.2 Four-phase Update

The large sneak currents in the 2-phase writing strategy, potentially resulting in false switching, is due to the high potential difference $V_P - V_{AP}$ between input terminals having different signs of inputs. One simple way to mitigate this issue is to further split the 2 phases of weight update so that, in a given phase, only rows having the same sign of input are updated at a time. This is equivalent to first clustering the columns according to the sign of δ, and then further clustering the rows according to the sign of x. This proposed 4-phase writing scheme would require additional transistors to choose the rows to be updated in a given phase as shown in fig. 4.6(c). It is summarized in Table 4.4 where each phase will have the same duration T_{wr}; thus the total time for updating the crossbar is doubled to $4T_{wr}$. Note that this is still $O(1)$ time. The required write voltages and transistor gate voltages can be obtained with very similar circuits as in fig. 4.5(a) and 4.5(b). The V_{saw} here would be same in phases 1 and 2, and the opposite in phases 3 and 4. In this scheme, the programming currents for each row remains as they were in the 2-phase update, just

	Input	Error	e_i	V_i^I	c_j	Switch
Phase 1	$x > 0$	$\delta > 0$	$u(x_i)V_{DD}$	$u(x_i)V_P$	$u(\delta_j)V_{DD}$	$P \rightarrow AP$
Phase 2	$x < 0$	$\delta > 0$	$u(-x_i)V_{DD}$	$u(-x_i)V_{AP}$	$u(\delta_j)V_{DD}$	$AP \rightarrow P$
Phase 3	$x > 0$	$\delta < 0$	$u(x_i)V_{DD}$	$u(x_i)V_{AP}$	$u(-\delta_j)V_{DD}$	$AP \rightarrow P$
Phase 4	$x < 0$	$\delta < 0$	$u(-x_i)V_{DD}$	$u(-x_i)V_P$	$u(-\delta_j)V_{DD}$	$P \rightarrow AP$

Table 4.4: 4-phase weight update for the 1R configuration in fig 4.6(c): Condition on input and error for a synapse to be updated, along with the control signals (e, c) and write voltages (V^I), for each phase

their time of appearance now differs. They still depend on the respective input and error.

Let us now see how bad the issue of sneak-path leakage is with this strategy. Fig 4.6(d) shows the equivalent circuit for the 2×2 crossbar with the same set of assumptions (only synapses providing alternate current paths are shown). For an $M \times N$ crossbar, in the worst-case scenario, sneak currents could be V_P/R_P and V_{AP}/R_{AP}, and can still result in false switching. This follows intuition as the potential difference between an input terminal and an output terminal is at most V_P or V_{AP}. However, in the average case, the sneak current values are found to be only $V_P/3R_P$ and $V_{AP}/3R_{AP}$. These currents are small, and do not have the potential to cause undesired switching as is evident from the parameters listed in table 4.3 and the range of values of V_P and V_{AP}. For eg. the soft switching threshold is about $45\mu A$ for $AP \rightarrow P$ switching with the maximum write pulse duration of $2.5ns$ (fig. 2.4 (b)), whereas the average case sneak current is $30\mu A$. Similarly, for $P \rightarrow AP$, the threshold is about $105\mu A$, while the average sneak current is $67\mu A$.

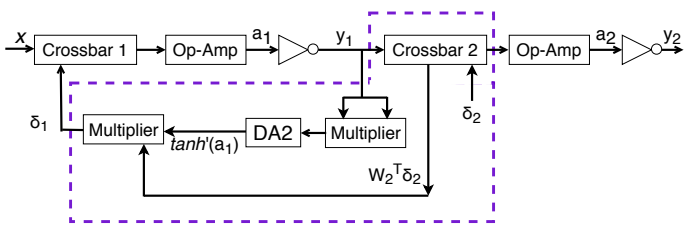
Hence, the 4-phase writing scheme significantly reduces the incidences of undesired switching at a small cost of increase in the duration of the write phase. As we shall see, this trade-off is not only worth but also necessary for satisfactory performance of the training process.

4.4.5 Multi-Layer NNs

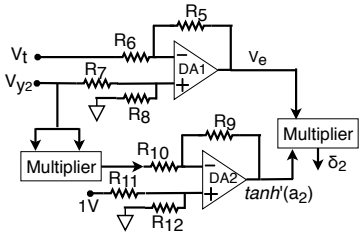
Multi-layer feed-forward NNs can be implemented on cascaded crossbars (each representing one layer) with the output of one fed as the input to the next. It is pretty straightforward to implement the backpropagation algorithm on such a structure, as demonstrated in fig. 4.7(a). Consider a 2-layer NN with weight matrices W_1 (hidden layer) and W_2 (output layer) represented by crossbars 1 and 2 respectively. For an input x, the final output y_2 is given as

$$y_2 = f(a_2) = f(W_2 y_1) \quad \text{where} \quad y_1 = f(a_1) = f(W_1 x) \tag{4.8}$$

The op-amp and inverter following crossbar 1 (just as in fig. 4.2(a)) compute y_1 which is provided as an input to crossbar 2. With a Mean Square Error cost function, the error of the 2^{nd} layer is given as $\delta_2 = 2(y_2 - t)f'(a_2)$, where t is the desired (target) value and f' denotes the derivative of activation function f. This is obtained as follows: y_2 and t, represented by V_{y_2} and V_t respectively, are fed to the inputs of a differential amplifier (DA1) as shown in fig. 4.7(b) to obtain the



(a) Error backpropagation requires 2 multipliers and a DA. The process involves the components within the dashed boundary.



(b) Computation of δ in the 2^{nd} layer

Figure 4.7: (a) Circuit for backpropagating errors to previous layers. (b). Circuit for finding the error at the last layer of the NN using the obtained value V_{y_2} and corresponding target V_t. In DA1, we need $R_5/R_6 = R_8/R_7 = 2$. For DA2, we use $R_9 = R_{10} = R_{11} = R_{12}$ to get $\tanh'(a_2) = 1 - \tanh^2(a_2)$.

difference $V_e = 2(V_{y_2} - V_t)$. We used *tanh* as the activation function f; its derivative is given as $\tanh'(x) = 1 - \tanh^2(x)$, which is obtained using a multiplier, such as the Hilbert multiplier [65], followed by the differential amplifier DA2. Lastly, the outputs from DA1 and DA2 are multiplied to get δ_2.

The error of the first (hidden) layer is given as $\delta_1 = ({W_2}^T \delta_2) \times f'(a_1)$, where $\times$ represents a component-wise product. As depicted in fig. 4.7(a), the matrix-vector product can be done on crossbar 2 itself by reversing the roles of its input and output terminals: δ_2 is now fed as the input and out comes ${W_2}^T \delta_2$, which, when multiplied by $f'(a_1)$, gives δ_1 as the error to be used for updating the weights of the hidden layer.

Note that DA1 is required only in the last layer of the NN to get the difference between the actual and target outputs. Whereas the components for backpropagation, comprising the 2 multipliers and DA2, are present in all layers and are a part of the Output CL. Also recall that the 2^{nd} layer error δ_2 has a dual role - deciding the MTJ write duration in crossbar 2 (with the circuit in fig. 4.5(b)), apart from being backpropagated to compute δ_1.

For the MTJ crossbar NN we described, during forward propagation, the total duration of the read phase would be at most nT_{rd} for an n-layer NN. Backpropagation of errors to hidden layers would require an extra T_{rd}-long read phase for each such layer, during which the error at (the output of) a layer is fed as an input to its crossbar to obtain the error at its preceding layer. Lastly, all the layers can be updated simultaneously (in $2T_{wr}$ or $4T_{wr}$ time, as per the architecture).

Further, it must be mentioned that a large layer in an NN could be split into multiple crossbars, some of which which share inputs or outputs. All these crossbars

can still be read and written in parallel, thanks to the locality of the weight update operations.

4.5 Training of Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) are a class of undirected graphical models used as generative models of data for the purpose of feature extraction, dimensionality reduction and classification [50]. They form the fundamental building block of Deep Belief Networks (DBNs) which have produced state-of-the-art results in learning tasks. In this section we provide a simplified mathematical background of RBMs and DBNs, and describe the in-situ training of RBM MTJ crossbars.

4.5.1 Basics of RBM

RBMs consist of a set of visible and hidden units to represent the data and their features respectively, and symmetric weighted connections between them as shown in fig. 4.8(a). The energy function of an RBM with visible and hidden units activations $\mathbf{v}$ and $\mathbf{h}$ and weights W is given as

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T W \mathbf{v} - a^T \mathbf{v} - b^T \mathbf{h} \tag{4.9}$$

where a and b are vectors of biases for the visible and hidden units. The conditional probability of the hidden units, given a certain state of the visible units, is

$$p(h_j = 1|\mathbf{v}) = \sigma(b_j + \sum_i W_{ij} v_i) \tag{4.10}$$

where $\sigma(x) = 1/(1 + exp(-x))$ is the logistic sigmoid function. Similarly,

$$p(v_i = 1|\mathbf{h}) = \sigma(a_i + \sum_j W_{ij} h_j) \tag{4.11}$$

The marginal probability of observing a certain visible vector $\mathbf{v}$ is computed as

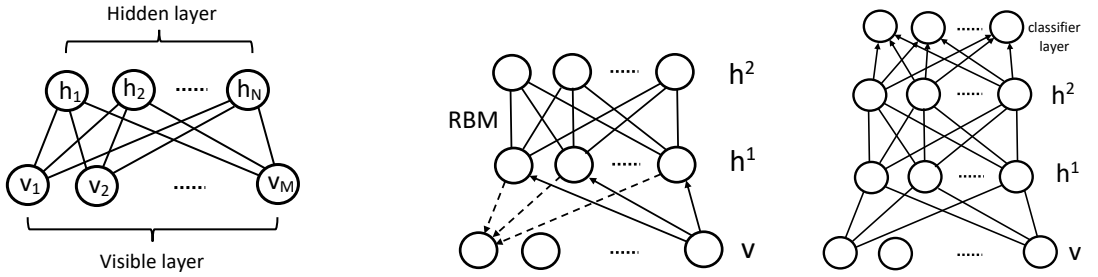
$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{4.12}$$

Training of an RBM involves maximizing the log likelihood of the probability of the training data and gives rise to a gradient ascent rule in the weight space. The weight update is calculated as

$$\Delta W_{ij} = \epsilon \frac{\partial \ log \ p(\mathbf{v})}{\partial W_{ij}} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{4.13}$$

where $\langle . \rangle$ denotes an expectation under the specified distribution and ϵ is the learning rate. Getting an unbiased sample of $\langle v_i h_j \rangle_{data}$ is simple because the absence of connections amongst the visible and hidden units lets us compute the probability with which the hidden units turn on using eqn. (4.10). However, getting an unbiased sample for the model is difficult because it requires starting from a random training vector and performing alternate Gibbs sampling for a long time, where each iteration of the sampling process updates the hidden states in parallel using eqn. (4.10) and then the visible states (again, in parallel) using eqn. (4.11). The mathematical model of RBM considered only binary states, but this has long been extended to include continuous values and model different kinds of data distributions [67].

In [49], a much faster learning method was proposed wherein the first (positive)



(a) An RBM with M visible and N hidden units.

(b) A DBN with 2 hidden layers

(c) A DBN with a classifier layer at the end

Figure 4.8: Schematics of (a) An RBM. The absence of connections within the visible and hidden layers makes the units in any layer conditionally independent of each other. All weights are symmetric. (b) A DBN. The top layer is an RBM being trained. In the lower layer, the dashed and solid arrows represent the top-down generative connections and bottom-up recognition connections respectively [50]. (Note: all weights are still symmetric; some connections removed for clarity.) (c) The DBN in (b) with a layer at the end for classification.

part of eqn. (4.13) is computed using the hidden units' activations obtained from eqn. (4.10). The second (negative) part is calculated by first reconstructing the visible units from the hidden states using eqn. (4.11) and then the hidden units from these reconstructed visible units (eqn. (4.10)). The weight update rule thus stands as

$$\Delta W_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon}) \tag{4.14}$$

This works well even though it only roughly approximates the gradient of the log probability of the training data, and is closer to the gradient of another objective function, the Contrastive Divergence (CD) [49]. Observe that this learning rule is also local, just like the one in eqn. (2.3). Further, since this CD weight update depends only on the training data and network parameters (the weights), and not on any labels, it comes under the category of unsupervised learning. The bias vectors a and b are also trained in a similar way.

One way to track the progress of learning is to measure the reconstruction

error, which is the squared difference between the training data and its reconstructed version [51].

4.5.2 Deep Belief Networks

RBMs can be stacked to form deep generative models called Deep Belief Networks(DBNs) [52] . The lower layers (which are close to the visible layer) capture low-level features, whereas the higher layers represent abstract concepts. Fig. 4.8 (b) illustrates a DBN with 2 hidden layers. In a DBN with l hidden layers, the joint distribution of the data $\mathbf{v}$ and the hidden layer variables $\mathbf{h}^1, \mathbf{h}^2, \ldots \mathbf{h}^l$ is expressed in terms of the conditional distributions [16].

$$P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots \mathbf{h}^l) = P(\mathbf{v}|\mathbf{h}^1)P(\mathbf{h}^1|\mathbf{h}^2)\ldots P(\mathbf{h}^{l-1}, \mathbf{h}^l) \tag{4.15}$$

Hinton et al. [52] have proposed a greedy layer-wise training procedure for the DBN, starting with the lowest hidden layer $\mathbf{h}^1$. Once it has been trained using the Contrastive Divergence formulation mentioned above, its weights are kept fixed and used to obtain the training data for the next layer $\mathbf{h}^2$. This is done by propagating the training samples $\mathbf{v}$ using the learned $P(\mathbf{h}^1|\mathbf{v})$ (computed using eqn. (4.10)), and using either these probability values or samples from their distribution as training data for the second layer. This is repeated for all subsequent hidden layers up to $\mathbf{h}^l$.

Often, DBNs are not used on their own; rather the features extracted by them are used for the purpose of classification by adding a classifier layer at the last hidden layer (fig. 4.8(c)) and using a supervised gradient descent algorithm to train the weights of this classifier. Another common practice is to use the weights of the

trained DBN for initializing the hidden layers of a deep feed-forward neural network. These weights are then fine-tuned with the supervised training criterion, along with the weights of the classifier layer(s) appended at the end. This unsupervised pre-training of the hidden layers, before the data labels are used, is justified on the grounds that random initialization of the weights of hidden layers often leads to the network getting stuck at local minima when using only supervised gradient descent methods. This is specially problematic for the lower layers as their activations tend to get saturated, leading to vanishing gradients which slow down the learning process [39].

4.5.3 Adaptation of the Contrastive Divergence algorithm

The standard CD algorithm comprises the following steps:

1. Clamp the visible nodes $\mathbf{v}$ to a training vector, say v_1.

2. Find the probabilities with which the hidden units turn on using eqn. (4.10). That is, compute $h_1^p = \sigma(Wv_1)$ (ignoring the bias for simplicity).

3. Obtain the binary states h_1^b of the hidden units by sampling from the probability distribution h_1^p. It is necessary to store the hidden states as binary values, rather than using the real-valued probabilities themselves, so that they can communicate a single-bit value during reconstruction, thereby acting as a strong regularizer [51]. This marks the end of a construction phase.

4. Reconstruct the states of the visible units using those of the hidden units just as in eqn. (4.11): $v_2^p = \sigma(W^T h_1^b)$. It is common to simply use these probability values as it reduces the sampling noise and hastens the learning process.

5. Now reconstruct the hidden units as $h_2^p = \sigma(W v_2^p)$. As per the recommendation in [51], it is *not* required to sample binary states from h_2^p so that unnecessary sampling noise can be avoided.

6. Perform the CD weight update as

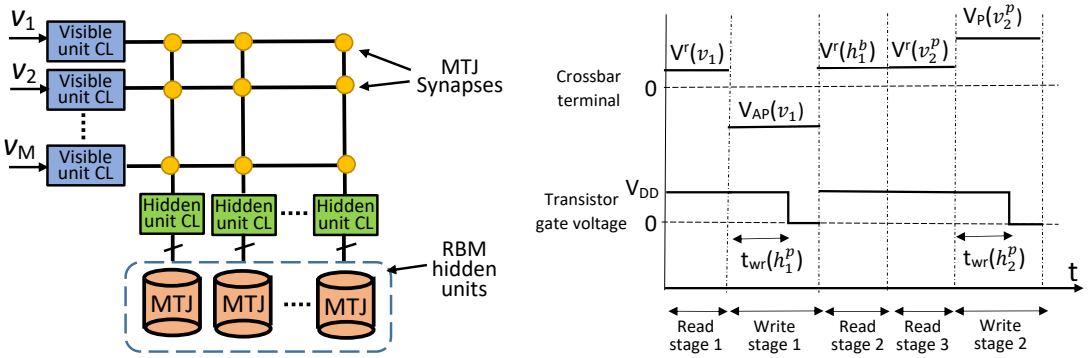
$$\Delta W = \epsilon(h_1^p v_1^T - h_2^p v_2^{pT}) \tag{4.16}$$

   For the data-driven positive part of the weight update, it is better to use h_1^p as it eliminates the sampling noise present in h_1^b. Note that eqn. (4.16) changes the weights with the statistics of only one training example, and thus doesn't require the expectation operator. The aforementioned steps should be repeated for all training samples over several iterations.

Now we go on to explain how the CD algorithm would be adapted for implementation on the MTJ crossbar. The standard CD algorithm has a weight update in eqn. (4.16) with 2 terms, each of which has activations of both the visible and hidden units. This makes it impossible to perform such a weight update on the crossbar without explicitly calculating and storing in memory at least the positive term. To avoid this, we choose to implement the updates from the construction and reconstruction phases separately. Further, since the v_1 and h_1^p are available at the end of step (2), the positive update can be done before the reconstruction. This further removes the necessity of storing v_1 and h_1^p while v_2^p and h_2^p are calculated. It has been observed that this 2-step weight update doesn't quite affect the RBM's learning [116]. We too shall verify this at a later stage.

In the construction phase, the binary states h_1^b of the hidden units are chosen by

sampling from the probabilities h_1^p (step (3)). Since the probabilities are generated from the output of a sigmoid activation function, and the MTJ switching behavior (fig. 2.4 (b)) bears close similarity to a sigmoidal response, we use an MTJ itself to produce and store the binary state of a hidden unit. The alternative to this would have been the use of some analog/digital random number generator to compare its output with h_1^p and generate a binary state; this is likely to have a higher overhead. We shall further discuss the implementation of this technique in the next subsection.



(a) RBM MTJ Crossbar and Hidden Layer

(b) Read and Write voltages

Figure 4.9: (a) Crossbar implementation structure of the RBM with MTJs as synapses and hidden units. (b) Signals during the 5 stages of the CD update cycle. The quantities on which they depend are in parentheses. The crossbar terminal voltages are at the visible unit CL for all except Read Stage 2, where the hidden units provide an input for reconstruction of visible units. All reads and writes are of duration T_{rd} and T_{wr} respectively.

4.5.4 Training of RBM MTJ crossbar

Fig. 4.9(a) depicts the RBM crossbar with Control Logic (CL) for each visible and hidden unit, and an MTJ for storing the binary state of the latter. The MTJ synapses could be with or without selection transistors. Because the reconstructed values of the visible units are outputs of the sigmoid and restricted to the range $(0, 1)$, we would require inputs to the RBM to be normalized to the same range for

better reconstruction. Each cycle of the CD algorithm implemented on the crossbar goes through 5 stages as listed below. The signals interfacing the crossbar are shown in fig. 4.9(b).

- Read Stage 1: The training starts with the crossbar visible terminals having a voltage $V^r \in [0, V_{rd}]$ proportional to the training input v_1. The current received at the hidden terminals would be used to flip the MTJ units storing the hidden states and simultaneously be converted to activations h_1^p.

- Write Stage 1: For the positive weight update stage, since both v_1 and h_1^p are positive, we would only require to switch the MTJ synapses from $AP \rightarrow P$ with a suitable probability. Just as in section 4.4.2, we linearly map v_1 and h_1^p to MTJ synaptic write current I_{wr} and pulse width t_{wr} respectively as

$$t_{wr} = t_0 + t_1 h_1^p \tag{4.17}$$

$$I_{wr} = I_0 + I_1 v_1 \tag{4.18}$$

  We use the same values of t_0, t_1, I_0 and I_1 as listed in table 4.3, which give write voltages $V_{AP} \in [-0.81, -0.62]$ as previously.

- Read Stage 2: The MTJs storing the binary states h_1^b of the hidden units are read and the hidden terminals are applied a voltage V^r depending on the value read. Since h_1^b is binary (0 or 1), V^r is either 0 or V_{rd}. The reconstruction of visible units v_2^p is obtained using the current flowing into the other end.

- Read Stage 3: A reconstruction of the hidden units (h_2^p) is obtained by feeding v_2^p to the crossbar. Unlike read stage 1, there is no need to sample binary states from

h_2^p.

- Write Stage 2: Lastly, the negative weight update, which would require MTJs to switch only from $P \rightarrow AP$, is carried out by passing currents with magnitude and duration proportional to v_2^p and h_2^p respectively, just as in eqn. 4.17 and 4.18. Only the polarity and current magnitudes are for $P \rightarrow AP$ switching; and $V_P \in [0.68, 0.98]$ volts.

The entire cycle thus takes $3T_{rd} + 2T_{wr}$ time. Since the logistic sigmoid σ is only a scaled and shifted version of $tanh$, the same circuit (that is, the inverter) can be used to realize it, although with different parameters such as R_f and V_{SS}. The hardware required to implement the proposed training algorithm is also pretty much the same as that in sec. 4.4.3.2, except that h replaces δ as the quantity that decides write time of MTJs, the inverter in the Output CL (fig. 4.5(b)) isn't required, and V_{saw} is the same in both Write stages.

In the training of the 1T1R NN crossbar in sec 4.4.3, the write stage had to be split into 2 because of the 4 possible combinations of the signs of the input and error. The RBM crossbar, however, doesn't require such splitting because the visible and hidden units' values driving the CD weight update are always positive, and weight updates of all synapses have the same sign in a given write stage.

The 1R crossbar in sec 4.4.4 had a 4-way split of the write stage because a 2-way split resulted in large sneak currents. On the other hand, an RBM crossbar with a 1R architecture would also have a single phase, for reasons same as those of the 1T1R crossbar. In any given write stage, all synapses are updated, which means all rows and columns are simultaneously active. Thus, transistors for selecting rows

(the ones labeled e_i in fig. 4.6(c)) are not required, and only columns would have selection transistors (labeled c_i) to control their respective write pulse widths t_{wr}. Since there are no sneak paths during writing, the scalability of 1R crossbar makes it the choice of architecture for an RBM.

At this stage one may ask why training inputs to the general NN crossbar should be bipolar, as was considered in sec. 2.3.1 and 4.4.1. The explanation lies in the faster convergence of the training when inputs are bipolar, or specifically have an average close to 0 [70]. If inputs x are normalized in $[0, 1]$, then the update of the weights connected to the j^{th} neuron ($x\delta_j$) would all have the same sign as that of error δ_j. Thus these weights would always move together, making the training process inefficient and slow [70].

Fig. 4.10(a) depicts 2 crossbars concatenated with each other forming a DBN with hidden layers h^1 and h^2. They would be trained sequentially using the procedure described above.

4.5.5 MTJs for hidden units

In Read Stage 1, the MTJ hidden units are provided with a switching current I_{sw} to switch them $AP \rightarrow P$ (say P state is 'on') with probability h_1^p. Their states are read in Read Stage 2 using a certain current I_{read}, and they are reset $P \rightarrow AP$ in either Read Stage 3 or Write Stage 3 in preparation for the next cycle. Fig. 4.10 (b) shows the circuit of the MTJ hidden units and table 4.5 summarizes its operation. The currents I_{sw} and I_{reset} flow in opposite directions to flip the MTJ from $AP \rightarrow P$ and $P \rightarrow AP$ respectively. The read current I_{read} could be in any direction.

We shall now provide a detailed description of how the stochastic switching

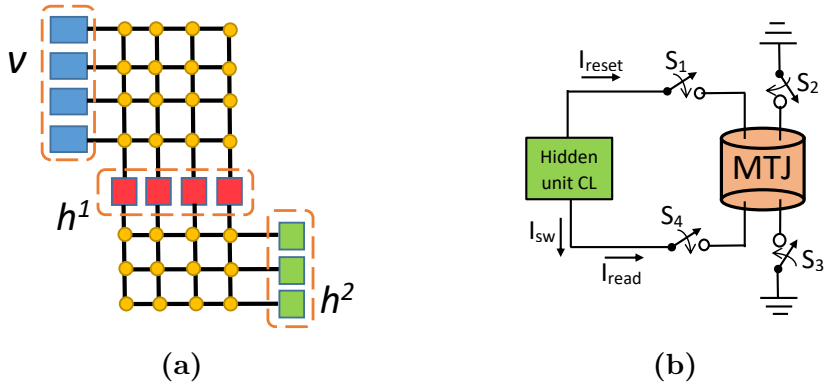


Figure 4.10: (a) DBN crossbar structure - 4×4 and 4×3 crossbars for the 1^{st} and 2^{nd} hidden layers. (b) Circuit of MTJ as RBM hidden unit connected to the respective Control Logic.

behavior of MTJs is used for sampling the binary states of the hidden units. Table 4.6 summarizes the notations adopted. We need to fit the transfer function of the sigmoid with the MTJ switching probability curve. A hardware-friendly method is a simple linear mapping of the incoming neuron current I_n to MTJ's switching current, which requires us to match the characteristics at exactly at two points - say for values of the weighted sum $a = 0$ and a_{cf} ('cf 'denotes curve fitting).

For the value of $a = 0$, we would have the neuron current $I_n = 0$. This should correspond to an MTJ switching current of I_0 and a probability of $\sigma(0) = 50\%$, that is, equal chances of the binary state to be 0 and 1. Recall from sec. 4.3.3 that MTJ conductances G_P and G_{AP} correspond to synaptic weights b and $-b$, and read

Current	Switch	Stage
I_{sw}	S_4, S_2	Read 1
I_{read}	S_4, S_1	Read 2
I_{reset}	S_1, S_3	Read 3 or Write 2

Table 4.5: The stages of CD training when different currents (fig. 4.10(b)) operate on the hidden units and the switches that are active

I_n	Neuron current at the hidden unit CL
I_{sw}	MTJ hidden unit switching current
a	Weighted sum input to sigmoid
I_0	Value of I_{sw} for $\sigma(0) = 0.5$ switching probability
I_{cf}	Value of I_{sw} for $\sigma(a_{cf})$ switching probability

Table 4.6: Notations for MTJ curve fitting

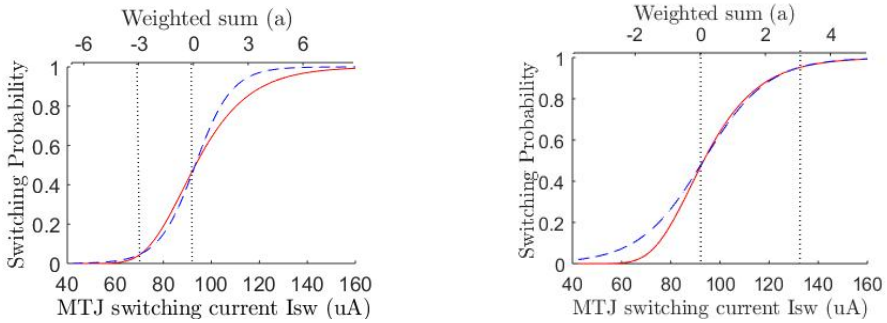
voltage V_{rd} to input of 1. Thus, for $a = a_{cf}$, we would have

$$I_n = a_{cf}\frac{V_{rd}(G_P - G_{AP})/2}{b} \tag{4.19}$$

and the MTJ switching current should be $I_{sw} = I_{cf}$. This gives the relation

$$I_{sw} = I_0 + \frac{2bI_n(I_{cf} - I_0)}{a_{cf}V_{rd}(G_P - G_{AP})} \tag{4.20}$$

which can be implemented using a differential amplifier. For our experiments we perform the curve fitting at $a_{cf} = -3$. The reason behind choosing a large value is to cover a significantly wide range of values of activations. Fig. 4.11(a) shows how close the MTJ switching probabilities are to the desired probabilities of activation as current I_{sw} and a are varied. The pulse width of I_{sw} has been chosen to be $2ns$, which is the duration of the read stages. In contrast, fig. 4.11(b) shows the same for $a_{cf} = 3$ wherein the probabilities match perfectly for positive values of a, but for $a < 0$ the MTJ switching probabilities obtained are significantly less than those of the desired values (that is, the transfer curve of the sigmoid). It is crucial to capture the small probabilities otherwise values of $a < -2$ would produce currents I_{sw} which are too small to ever flip the MTJs and turn the hidden units 'on'. This will cause them to convey $h_1^b = 0$ in the reconstruction phase (step (4)) much more frequently than they should. Also, one may consider not matching the curves at 0, and instead match at $a_{cf} = 3$ and -3; however, this results in poor fitting in intermediate values, with difference in probabilities being higher than 0.15 for a wide range of values of a.



(a) With $a_{cf} = -3$ **(b)** With $a_{cf} = 3$

Figure 4.11: Switching probabilities of the MTJ (solid red line) meant to store RBM hidden unit's states as a function of I_{sw}, and desired hidden unit's activations as output of sigmoid (dashed blue line) for (a) $a_{cf} = -3$, and (b) $a_{cf} = 3$. Vertical dashed lines in the plots depict the matching at respective values of a_{cf}. In (a), the parameters are $I_0 = 93.28\mu A$, $I_{cf} = 70.38\mu A$. Maximum read voltage $V_{rd} = 0.2\ V$ has been used throughout.

4.6 Simulation Setup and Results

To see how successfully the MTJ crossbar NNs and RBMs[1] can be trained in-situ, we performed system level simulations by modeling the functionality of the crossbar architecture in MATLAB and training it on some datasets. To capture the MTJ device parameters, we used an HSPICE model [60] and included thermal fields in its LLG equations for obtaining the stochastic switching characteristics [111]. Certain device parameters used in and obtained from this model[2] were then incorporated into the simulations of the crossbar. We discuss the results obtained on feed-forward NNs and DBNs in that order.

[1] A tutorial on DBNs, along with code in Theano, can be found in [5] and [10]

[2] MTJ cell dimensions - $35nm \times 35nm \times 1.4nm$, $R_P = 4.86k\Omega, R_{AP} = 15.12k\Omega$, temperature $T = 300K$, saturation magnetization $M_s = 1029emu/cm^3$, damping constant $\alpha = 0.014$ yielded $\Delta = 40$, $I_{c0}^{P\rightarrow AP} = 64.5\mu A$, $I_{c0}^{AP\rightarrow P} = 21.2\mu A$

4.6.1 Neural Networks

4.6.1.1 Methodology

The performance of the NN was evaluated in the following scenarios (code-named for further reference). All training processes used the Mean Square Error cost function and neurons had the *tanh* activation function.

1. **RV:** First, we train and evaluate an NN with real-valued weights in MATLAB. Binary quantization step (b) is obtained from this trained network as shown in sec. 4.3.3.
2. **DP:** Suitable binary weights are obtained by doing probabilistic learning in software on a binary network. Then a 1T1R crossbar and a 1R crossbar are deterministically programmed to these weights. We see the effect of device variations on the former, and of alternate current paths and resulting false switchings on the latter.
3. **ST:** An MTJ synaptic crossbar is modeled and stochastically trained in-situ using the linear model of stochastic weight update described in sec. 4.4.2 for the
   (a) 1T1R architecture, with the 2-phase write strategy (sec. 4.4.3).
   (b) 1R architecture, with both the 2-phase (to see the effects of sneak currents) and the 4-phase update strategies (sec. 4.4.4). For the former, node voltages of output terminals not connected to the output CLs (that is, columns not being updated) could be easily calculated using Kirchhoff's Current and Voltage laws. Whereas for the latter, a mesh analysis of the crossbar was required and node voltages at both (unconnected) input and output

terminals were obtained by solving a system of linear equations of KCL and KVL in MATLAB.

4. **DV:** Device variations of different extent are introduced in the stochastic training of both the 1T1R and 1R crossbars. It reflects in the variations in the resistance of the P and AP states, the standard deviations of which usually do not exceed 10% of their mean values as per experiments [136].

We use the following datasets for evaluation.

SONAR, Rocks vs Mines[78]: Three different NN architectures are considered - one with 1 layer (1L), and two with 2 layers having 15 and 25 hidden neurons respectively, and named 2L15 and 2L25. They were trained, and then tested on 104 samples of the test dataset.

MNIST Digit Recognition[68]: Three 2-layer networks of 50, 100 and 150 hidden units respectively and a 3-layer network of 50+25 hidden units were trained on the first 10000 samples of the training set and then evaluated on the 10000 images of the test dataset.

Wisconsin Breast Cancer (Diagnostic)(WBCD)[78]: A single-layer network (1L) and 2 two-layer networks (2L10 and 2L20) were considered, and the test dataset had 200 samples.

4.6.1.2 Results

Table 4.7 summarizes the accuracy obtained with these networks under the different training scenarios mentioned above. The effect of device variations of different extents on the in-situ stochastic training is highlighted for some of the networks in table 4.8, with fig. 4.12 plotting the mean square error as the training progresses for

Dataset		SONAR			MNIST				WBCD		
Network		1L	2L15	2L25	2L50	2L100	2L150	3L	1L	2L10	2L20
RV		16.4	12.8	11.9	9.87	7.34	6.44	7.25	8.35	7.40	7.10
DP	1T1R	19.2	15.2	14.3	13.50	10.89	9.55	10.45	9.85	8.30	8.55
	1R	46.8	41.4	42.7	39.42	36.10	37.92	40.48	24.95	27.60	23.65
ST	1T1R	18.4	14.2	13.6	12.69	10.18	8.96	9.71	9.20	7.70	8.05
	1R	18.3	14.5	14.0	12.72	10.20	9.03	9.66	9.40	7.85	7.95

Table 4.7: Classification error rates for the 3 datasets (on the test samples) with various NN and crossbar architectures under different training scenarios. Here, ST-1R crossbar used 4-phase update. Ideal devices assumed for all except DP-1T1R, where 10% variation was considered. SONAR and WBCD figures are average of 10 runs. MNIST and WBCD figures are in %

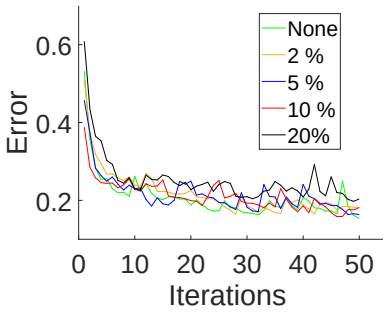
Dataset	SONAR				MNIST				WBCD	
Network	1L		2L15		2L100		3L		2L20	
Variation	1T1R	1R	1T1R	1R	1T1R	1R	1T1R	1R	1T1R	1R
2%	18.5	18.4	14.4	14.7	10.27	10.22	9.67	9.73	8.10	8.05
5%	18.7	18.7	14.7	14.8	10.28	10.29	9.78	9.80	8.25	8.30
10%	19.0	19.1	15.1	15.1	10.33	10.43	9.86	9.91	8.30	8.40
20%	19.3	19.5	16.0	15.9	10.42	10.72	10.15	10.28	8.60	8.75

Table 4.8: Misclassification rates of NNs with stochastic training (ST) of 1T1R and 1R architectures under different levels of device variations (DV) expressed in terms of standard deviations of R_P and R_{AP} about their mean values. MNIST figures are worst of 3 runs, SONAR & WBCD are average of 10 worst runs.

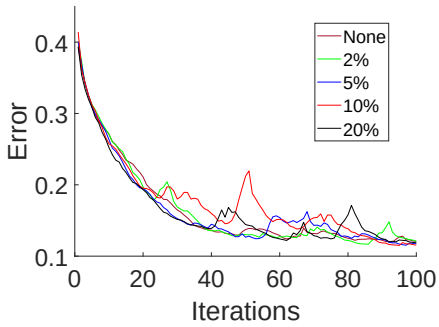
the 1R crossbar. Additionally, fig. 4.13 compares the error for the two write strategies. It doesn't converge with the 2-phase writing scheme due to higher instances of undesired weight changes, but does so with 4 phases.

It is evident from these results that

- When an MTJ synaptic crossbar without access transistors is stochastically trained in-situ (ST-1R), it shows classification accuracy only slightly lower (about 3% at worst) than when the same network is trained in software with real-valued weights (RV, which can be considered to be the best achievable). However, it brings about significant improvement (up to 30%) in accuracy over a deterministically programmed crossbar (DP-1R) since the latter suffers from undesired weight changes arising from alternate current paths.
- In-situ training also benefits the crossbar with transistors (ST-1T1R against DP-

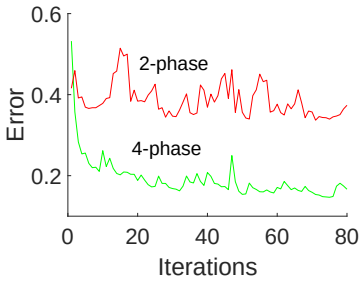


(a) On SONAR for 2L15 network

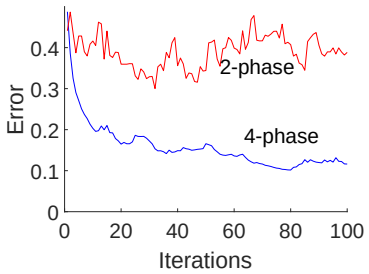


(b) On MNIST for the 3L network

Figure 4.12: NN training error with different extents of device variations on the 1R crossbar for 2 datasets.



(a) On SONAR for 2L15 network



(b) On MNIST for 2L100 network

Figure 4.13: Comparison of error during training of the 1R crossbar with 2-phase and 4-phase update schemes for 2 datasets. No variations assumed.

1T1R) in the presence of device variations by slightly improving accuracy (by about $0.5\% - 1\%$).

- It is possible to compensate for the loss in accuracy due to use of a binary network by increasing the size of the network (adding more hidden layers and/or neurons).
- Further, the trained crossbar has robustness even in the face of device variations, owing primarily to the fault-tolerant nature of NN and its learning algorithms. As can be seen in table 4.8, increase in misclassification rates remain within 2% even with 20% variation.

The accuracy degradation of $2\% - 3\%$ that we achieve (on going from RV to ST) is comparable to the 3.73% reported by [142] and the $0.8\% - 3.5\%$ in [130].

However, it must be mentioned and emphasized that any comparison is fair only if they are on the same dataset and network architecture. The benefit of using in-situ training can also be seen when we compare our work with that of [141] (which performs offline learning). On the MNIST 2L100 network, we obtained an error rate of 10.20%, whereas [141] had a much higher value of 30% on the same network, although it must be mentioned that the latter were at a disadvantage due to linear activation units. Further, the presence of a 20% device variability reduces our accuracy by less than 1.5%, which is competitive with that of [142],[141] and [130].

There are a few similarities and differences between our work and the MTJ synapse-based STDP learning proposed in [119] that we would like to first mention. In our work, all MTJ synapses from an input share the circuit that decides programming currents, and all synapses to an output neuron have the same programming duration. Similarly, in [119], the STDP learning circuit for synaptic potentiation (and depression) are shared by the synapses that connect an input neuron (pre-neuron) to all excitatory neurons. However, they get programmed with different currents depending on their respective post-neuron spiking time, but their write durations are always the same and independent of any spike times; it is only the write current which varies from synapse to synapse. The STDP learning circuit consists only of 2 sets of 2 transistors and a capacitor. On the other hand, our implementation of stochastic learning would require more complex hardware (primarily 2 op-amps in the Input CLs and 1 op-amp in the Output CLs). Also, we need multiplier circuits [65] to back-propagate errors to hidden layers of the network.

However, Srinivasan et. al. [119] have not described the hardware implemen-

tation of the neurons of the SNN and their functionality, although one possible configuration appears in [110]. This neuron in [110], while not being very complicated, is seemingly more expensive than an op-amp [65] in terms of area requirements. But overall complexity is perhaps higher for our design. On the other hand, [119] achieves only about 75% classification accuracy on the MNIST dataset on an SNN with as many as 400 excitatory and 400 inhibitory neurons trained with 460 images. We could get higher accuracies on our MNIST networks, although we trained with many more (10000) images.

4.6.2 Deep Belief Networks

For the training of the RBM crossbar, we consider only a 1R architecture since the absence of sneak currents (as discussed in sec. 4.5.4) does not leave any difference in the training procedure of the 1T1R and 1R crossbars. The performance of the DBNs was evaluated in scenarios similar to sec. 4.6.1 - first with real-valued weights (RV), then deterministically programming (DP) the MTJ crossbar to suitable binary weights, and finally performing Stochastic Training (ST) of crossbar without and with various extents of device variations (DV). Two datasets were used for obtaining data:

- **MNIST**: Two 2-layered networks with 150 and 200 hidden units, and two 3-layered networks with 150 and 200 units in each hidden layer were trained on the first 10000 samples of the training set and then evaluated on all test samples.
- **WBCD**: One 2-layered network of 40 hidden units was considered.

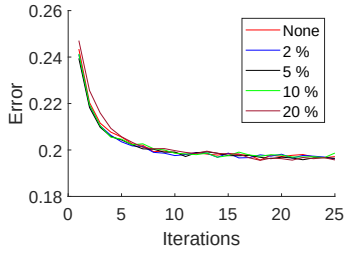
The last (output) layer of all networks was used for classification purposes, either using only the features extracted (abbreviated FE) at the last hidden layer as

Dataset	MNIST								WBCD	
Network Size	150		150 + 150		200		200 + 200		40	
Purpose	FE	SFT	FE	SFT	FE	SFT	FE	SFT	FE	SFT
RV	8.85	6.63	8.07	5.12	8.73	5.30	7.98	4.27	7.90	1.30
DP	38.29	34.40	37.83	41.54	39.17	36.53	37.92	40.07	24.00	29.40
ST	12.72	8.82	11.74	8.23	12.69	8.09	11.40	7.08	11.60	4.10
DV - 2%	12.93	8.97	11.77	8.33	12.75	8.21	11.58	7.19	11.80	4.20
DV - 5%	13.05	8.96	11.89	8.55	12.84	8.34	11.76	7.27	12.20	4.50
DV - 10%	13.22	9.18	12.15	8.70	13.02	8.71	12.00	7.34	12.50	4.50
DV - 20%	13.56	9.29	12.44	9.12	13.35	8.87	12.39	7.72	12.60	4.70

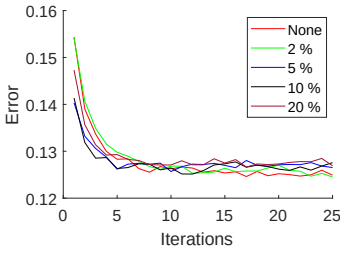
Table 4.9: Misclassification rates with hidden layers trained as RBM for the MNIST and WBCD datasets, for different levels of device variations. For the figures reported with $2\% - 20\%$ variations, the ones for MNIST are worst of 3 runs and those for WBCD are average of 5 worst runs out of 10.

classifier inputs, or fine-tuning (SFT) the hidden layer weights, along with training of the output layer, with the supervised training method used in neural networks (refer sec. 4.5.2). Table 4.9 lists the classification error rates obtained with all networks and training scenarios. As is clearly evident, it remains within $4-5\%$ and $3-3.5\%$ for FE and SFT respectively even with high levels of variations. Fig. 4.14 depicts how the different kinds of errors converge both with and without variations. For MNIST, plots of only the DBN with $200 + 200$ hidden units are shown.

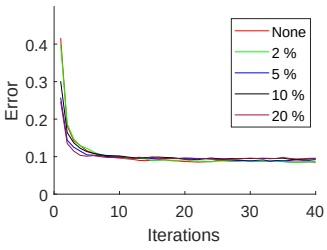
Additionally, fig. 4.15(a) and (b) compare the training with the standard CD algorithm and the 2-step CD that we use to train the MTJ crossbars, as described in sec. 4.5.3 and 4.5.4 respectively. Real continuous weights were used for the sake of this comparison on the MNIST 200 + 200 network. Both the reconstruction errors and the classification MSE of the 2 different implementations of CD are barely distinguishable. Lastly, fig. 4.15(c) depicts the bias of the weighted sum inputs of the hidden layer towards negative values, which justifies the tight curve fitting for $a < 0$ done in fig. 4.11(a). Apparently, the reason for this bias was the average input value (across all input units and data samples) being less than 0.5 for both



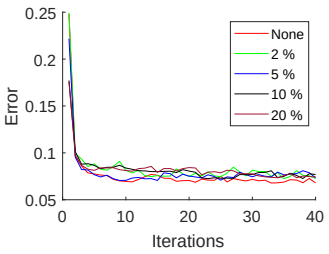
(a) Reconstruction Error Layer 1



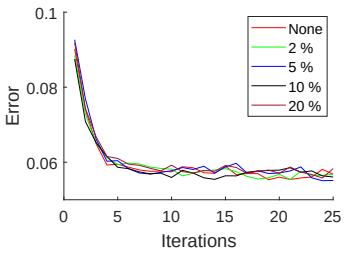
(b) Reconstruction Error Layer 2



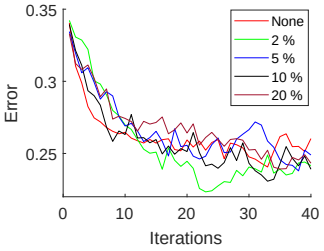
(c) MSE Feature Extraction



(d) MSE Supervised Fine Tuning



(e) Reconstruction Error

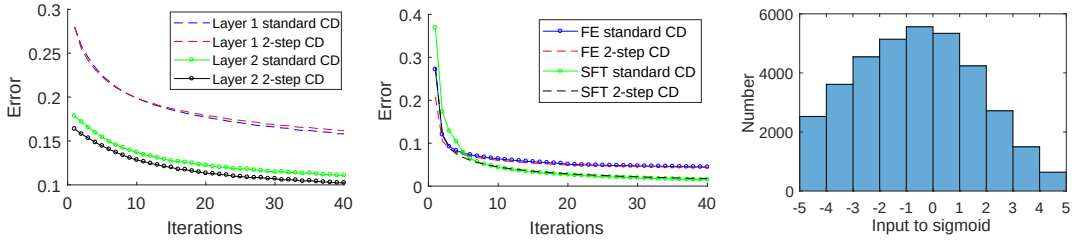


(f) MSE Supervised Fine Tuning

Figure 4.14: Progress of training with RBMs as hidden layers for (a)-(d) MNIST dataset, with 2 hidden layers each of 200 units, and (e)-(f) WBCD dataset, 1 hidden layer with 40 units. (a) and (b) show the reconstruction error on the 1^{st} and 2^{nd} layers respectively. (c) and (d) are the classification Mean Square Error with FE and SFT.

MNIST and WBCD. This required a reconstruction value (of visible units) of < 0.5 for low errors, which tend to shift the weights to negative values during the learning process so as to obtain negative weighted sums on an average. Other datasets with different characteristics may be suited to a different fitting (such as the one in fig. 4.11(b)) which can be easily done using techniques described in sec. 4.5.5.

In [94], a spiking neuromorphic system trained with event-driven CD was used for learning MNIST digits. The architecture had additional neurons in the visible layer for class labels (since the RBM was discriminative [67]), and the weights connecting the 500 hidden neurons to these class neurons were also trained using CD. Their model had a recognition error of 8.1%. The hybrid RRAM-CMOS RBM architecture of [120] obtains an average error rate of about 11% with 100 neurons in the hidden layers and a separate classification layer (which is similar to our approach,



(a) Reconstruction Error Layer 1 & 2 **(b)** MSE - FE and SFT **(c)** Distribution of weighted sums

Figure 4.15: (a) and (b) Comparison of standard CD and 2-stage CD in terms of how the reconstruction errors (a) and classification MSE (b) converge with iterations of training. The small gap between the green and black dotted lines (representing standard and 2-step CD resp.) is only due to the different initializations of the weights of the 2^{nd} hidden layer. Importantly, this gap remains same throughout, indicating same rate of convergence. (c) Histogram showing distribution of inputs to the sigmoid activation function of the 1^{st} hidden layer. All plots on MNIST $200 + 200$ network.

and unlike that of [94]). Lastly, the memristor-based RBM in [116], with 500 hidden neurons and 40 additional visible nodes, classifies 87.55% MNIST digits correctly but achieves convergence within only 5 epochs of 10000 training samples.

4.7 Discussion

We now analyze several other aspects of the in-situ training method proposed by us.

- **Training Time:** Training of an n-layer neural network on 1 1T1R crossbar using gradient descent will take $(2n-1)T_{rd} + 2T_{wr}$ time per training sample per iteration, as per sec. 4.4.5. On the other hand, consider a DBN with $(n-1)$ hidden layers and a final classification layer. We saw that the 2-step CD algorithm takes $3T_{rd} + 2T_{wr}$ per sample for an RBM. A DBN would be trained layer-wise where the training data for the r^{th} layer would be first obtained by propagating the original training sample through the preceding $r-1$ hidden layers, assuming that there is no storage of data in on-chip or off-chip memory. Thereby, the total

time for the r^{th} layer is $(r+2)T_{rd} + 2T_{wr}$; summing over r from 1 to $(n-1)$ gives a quadratic dependence on n for duration of training of the hidden layers. After this unsupervised learning, training of the classifier layer through gradient descent would take $nT_{rd} + 2T_{wr}$ if only the features extracted by the last hidden layer are used, whereas if we go for supervised fine-tuning of the entire network, it's $(2n-1)T_{rd} + 2T_{wr}$.

The higher time requirement for DBN training may be justified by the relatively smaller number of training iterations (typically $10-20$) within which the reconstruction error and MSE converge as compared to the larger number of iterations required if the network is trained entirely in a supervised way (compare fig. 4.14 with fig. 4.12). If in the DBN, the training data for subsequent hidden layers is stored instead of calculated, then memory can be traded-off for a linear dependence of training time.

- **Power consumption in the crossbar:** Let us estimate the expected power dissipated in the 1T1R MTJ crossbar NN by assuming an average case for all parameters. All inputs x and δ are half of maximum, that is ± 0.5. Thus, read voltages are half of maximum, that is, $V_{rd}/2$ and write voltage are those for $x = 0.5$, that is, $V_P(0.5)$ or $V_{AP}(0.5)$, denoted $\overline{V_P}$ or $\overline{V_{AP}}$. At all times, half of the MTJs are considered to be in the P state, and the rest half in AP state. This gives an average power in the read phase per MTJ synapse to be

$$P_{rd} = \frac{1}{2}(V_{rd}/2)^2 \left(\frac{1}{R_P} + \frac{1}{R_{AP}}\right) \tag{4.21}$$

and that in each of the 2 write phases to be

$$P_{wr} = \frac{1}{8}(\overline{V_P}^2 + \overline{V_{AP}}^2)(\frac{1}{R_P} + \frac{1}{R_{AP}}) \tag{4.22}$$

This yield the average power per device in a cycle of training to be $P_{rd} + 2P_{wr}$. Substituting values stated previously, this calculates to 82 μW. Taking $T_{rd} = 2ns$ and an average $t_{wr} = 2ns$, the energy consumed per device per cycle is 0.164 pJ. The 1R crossbar NN without transistors would have higher energy dissipation due to sneak currents. It must be noted that these values are heavily dependent on device parameters. Future MTJ technologies with scaled down devices would consume lesser energy.

For the RBM crossbar, P_{rd} remains the same. Write stages 1 and 2 have average write voltages $\overline{V_{AP}}$ and $\overline{V_P}$ respectively, and current flows through all synapses in both stages. Thus average power per synapse per cycle is $3P_{rd} + \frac{1}{2}(\overline{V_P}^2 + \overline{V_{AP}}^2)(\frac{1}{R_P} + \frac{1}{R_{AP}})$, which turns out to be 169μW, and the average energy is then 0.338 pJ.

- One very popular work with binary weights is BinaryConnect [30] wherein the weights used during the forward and backward propagation are binary and obtained stochastically from real-valued weights. However, the weight update step is not binarized to maintain a good precision of the weights, as in the updates are real-valued. The performance of BinaryConnect is reported to be as good as, or even better than, their counterparts with continuous weights. However, the MTJ crossbar (or any binary device weight array) would not allow for storing of real-valued weights, which perhaps explains a noticeable, though not significant,

drop in classification accuracy when compared with floating-point weights.

- A drawback of in-situ training is that every chip has to be trained separately, each requiring roughly the same amount of time. Also, only the training algorithm for which the chip is designed (for eg. CD) can be used, unless extra hardware is added for the implementation of different techniques [139].
- **Dependence on temperature**: Higher operating temperatures reduces the thermal stability of the MTJs ($\Delta \propto 1/T$) and increases the switching probability for the same current magnitude and duration. The curves in fig. 2.4 shift to the left.
- Binary nature of MTJs severely limits the precision of each synaptic weight, thereby requiring larger crossbars with more hidden units to reach the accuracy exhibited by real continuous weights. On the other hand, while memristive devices do have several intermediate states, it's often difficult to program them reliably; so they too may end up being used in binary mode [96]. Further advances in materials of both magnetic and memristive devices will improve their prospects for use in memory and logic units.

4.8 Conclusion

In this work, we show how MTJ crossbars representing weights of ANNs can be trained in-situ by exploiting the stochastic switching properties of MTJs and performing weight updates in a way akin to gradient descent. We demonstrate how the machine learning algorithm can be implemented on crossbars with and without transistors. Results show these stochastically trained binary networks can achieve classification accuracy almost as good as that of those trained in software and im-

plemented on processors. This paves the way for the attainment of highly scalable neural systems in the future capable of performing complex applications.

Chapter 5: MTJ-based Ising Model Architecture

While the last 2 chapters have focused on the implementation of Neural Networks using MTJs in non-conventional and non-von Neumann computing paradigms, this chapter turns attention towards another computationally intensive type of workloads - combinatorial optimization problems. The parallelism offered by non-von Neumann architectures opens up a new path for finding good local optimum of such intractable problems.

5.1 Introduction and Related Work

Several real world problems come under the category of combinatorial optimization and are NP-hard, for eg. the travelling salesman problem, graph coloring, etc. This means that the problems are not computationally scalable with traditional von Neumann computing methods [95]. The capabilities provided by non-von Neumann architectures have motivated research [122, 28, 44] on accelerating the process of finding local optimum of such problems.

The Ising model [26], a mathematical framework to describe interactions between magnetic spins, can be leveraged to express and formulate many NP-hard problems due to the combinatorial nature of the model. It consists of a system of spins which can take one of 2 possible values $\{1, -1\}$. These spins interact with one

another in such a way that the system gradually evolves to a minimum energy state, representing a solution to the NP-hard problem that it encodes.

5.1.1 Related Work

The computational complexity of the Ising model has long been explored and investigated, and so has been the search for efficient hardware systems [13, 140, 85, 66, 18, 19] for mapping combinatorial problems. For example, the process of quantum annealing [66, 8] naturally holds the capability to tackle problems encoded as the Ising model, which requires the system to move out of local minima so as to continue converging to the ground state. However, the quantum technology is far from reaching maturity in terms of a large-scale commercial use due to its requirement of operating superconducting devices at very low temperatures. CMOS-based implementations [140] of Ising solvers have also been looked at, including the use of GPUs [27] for exploiting the inherent parallelism of Ising computations. However, some of these have made use of extra hardware [43, 85] or memory [27] for generating random numbers to simulate annealing properties in the model. Further, the Ising model often requires a large number of connections among Ising spins, which has led to the use of techniques such as cell cloning in fixed 2-D spin arrays [43], or to retaining only the nearest neighbor connections [140] leading to sub-optimal outcomes.

Recent work [117, 115, 122] has investigated the use of spintronic (nanomagnetic) devices for emulating the behavior of Ising spins by exploiting their natural physics. The work in [122] demonstrates through simulations such capability in stochastic nanomagnets operating at very high speeds; but these had very low en-

ergy barriers, implying that in reality they can suffer from fabrication complexity, read disturbs, and inability to write to several other Ising spins. Shim et al. [117] have used Magnetic Tunnel Junctions (MTJs) with higher energy barriers as Ising spin devices. However, they limit Ising spin connectivity to only the (four) nearest neighbors, and restrict their interactions to binary. Although this strategy yields a simple design, it severely limits the nature and size of NP-hard problems that can be encoded onto the hardware. The work in [115] does not detail how the influences from different units, in the form of voltages, would be added up.

5.1.2 Our contribution

In our work, we propose to evaluate an Ising model computing platform based on stable MTJs. We aim to tackle simultaneously several of the aforementioned issues not addressed in previous spintronic-based works. Our contributions are as follows:

- We design the hardware of an Ising cell, where an MTJ represents an Ising unit, and show how it can perform Ising computations.
- We demonstrate how a cell with a fixed no. of inputs can be slightly modified to make it scalable to large problems.
- We then propose Ising-FPGA, a parallel and reconfigurable architecture composed of several of these Ising cells, and having an interconnect topology similar to an FPGA.
- We analyze the degradation in signals in the hardware platform to get a more realistic picture of such implementations, and attempt to take them into account while mapping an NP-hard problem.

5.2 The Ising Model

The Ising model was originally developed to study the behavior of ferromagnets and consists of a number of spin units (ferromagnetic elements) with pairwise interactions [26]. The energy of the system is described by the Ising Hamiltonian

$$H(x) = -\sum_{i,j}^{N} J_{ij} x_i x_j - \sum_{i}^{N} h_i x_i \tag{5.1}$$

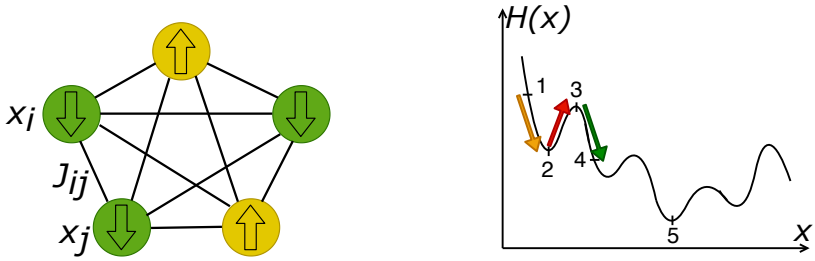
where N is the no. of units, x_i is the spin of the i^{th} unit and can assume one of 2 values, say '+1' (up spin) and '−1' (down spin), J_{ij} is the coefficient of pairwise interaction between the i^{th} and the j^{th} units, and h_i is a bias term accounting for external fields. Fig. 5.1(a) shows a complete Ising graph with 5 units. Note that the model considers a symmetric J, implying a reciprocal nature of the interactions. Also, there are no self-interactions, thus $J_{ii} = 0$.

Solving the system involves finding a configuration x of the spin units that minimizes the energy H. Obtaining this ground state is an NP-hard problem due to the discrete nature of x_i, and this property of the Ising model has enabled the mapping of several combinatorial optimization problems to it [82]. The ground state of the spins represents the solution of the NP-hard problem it encodes.

Theoretically, the probability of finding the system in a particular state x is given as [26]

$$P(x) = \frac{e^{-H(x)/(k_B T)}}{\sum_y e^{-H(y)/(k_B T)}} \tag{5.2}$$

where k_B is the Boltzmann constant and T is the temperature of the system. At high temperatures, the system explores the solution space and is almost equally



(a) A general Ising graph **(b)** Ising Hamiltonian

Figure 5.1: (a) An Ising graph with 5 spin units. (b) The system transitions from state 1 to 2, a local minima. Random perturbations can take it to state 3 so that it transitions to 4 and can eventually reach global optimum 5.

likely to be found in any state [26]. Whereas at low temperatures, states with lower energy would dominate. Ideally, the system should start from a high temperature and be slowly cooled down - a process known as annealing - so that it eventually reaches the ground state.

The underlying parallelism in the model can be exploited while searching for the ground state. The energy due to a single unit x_i and its connections, called the local Hamiltonian, is expressed as [27]

$$H(x_i) = -\sum_{j}^{N} J_{ij}x_j x_i - h_i x_i \tag{5.3}$$

which considers the interactions with its neighbors and its bias. Each step in the process of finding the ground state of the system involves lowering the local Hamiltonian of each unit *in parallel* which can be done by simply changing the state of x_i if that helps lower $H(x_i)$. However, the system would soon get stuck in a local minima rather than converging to the global optimum. The way out of this is to randomly perturb the system and allow it to go to a higher energy state for the time being - a popular concept known as (simulated) annealing. Fig. 5.1(b) depicts the energy landscape with the local and global minima, and demonstrates the effect of

annealing.

5.3 Ising-FPGA Framework

An NP-hard problem with N variables requires N Ising units, implying an $O(N^2)$ connectivity among the units. Also, the specific nature/type of the connections depends on the problem itself. We therefore envision a reconfigurable MTJ-based architecture which allows a large class of Ising models to be implemented. To this end, we leverage the advancements made in the FPGA technology to propose a similar architecture for our Ising-model hardware platform, and call it the *Ising-FPGA*. In this chapter, we present the design of such an MTJ-based Ising-FPGA possessing a routing network similar to regular FPGAs. We estimate the effects of the hardware platform on the Ising model computations and solution quality, and develop techniques which account for or mitigate them.

It must be noted that the Ising-FPGA is only an architecture, consisting of an array of MTJs, which exhibits reconfigurability and has a routing topology similar to FPGAs. It serves the purpose of mapping problems which can be formulated using the Ising model. The Ising-FPGA is *not* a standard FPGA, with some components are made of MTJs, and which is to be used for mapping digital logic functions.

5.3.1 Finding local optimum in the Ising model

The local Hamiltonian in eqn. 5.3 tells us how the spin of an Ising unit should be modified towards lower energy. Taking the negative of derivative of both sides, we

get

$$-\frac{\partial H(x_i)}{\partial x_i} = \sum_{j}^{N} J_{ij}x_j + h_i = \beta_i \ (say) \tag{5.4}$$

where β_i represents the cumulative *influence* on the i^{th} unit by the other units (all x_j). The sign of β_i at a certain time step decides the direction in which x_i should be updated to lower the local energy. For eg. if $x_i = -1$, and $\beta_i > 0$, x_i should be switched to $+1$ (otherwise it should remain at -1). This is similar to a gradient descent approach, although it must be noted that x_i can only be binary.

Algorithm 1 summarizes the general process involving the Ising model. After all spin units are initialized randomly (line 1), each iteration involves calculating influence β_i (line 4), modifying the spin value accordingly (line 7), and then flipping it randomly with a small probability (line 8) to enable escaping from local minima (fig. 5.1(b)). Observe that both the inner **for** loops can be executed in parallel for the N units.

Algorithm 1 Annealing process for the Ising model

1: Initialize all x_i randomly from $\{-1, 1\}$
2: **for** $n = 1$ to $iters$ **do**
3: **for** $i = 1$ to N **do**
4: Calculate β_i from eqn. 5.4
5: **end for**
6: **for** $i = 1$ to N **do**
7: $x'_i = sign(\beta_i)$
8: $x'_i = -x'_i$ with probability $p << 1$
9: **end for**
10: Assign $x = x'$ and reduce p.
11: **end for**

5.3.2 MTJ as an Ising spin unit

The stochastic switching characteristics of the MTJ has been an impediment to the realization of energy-efficient STT-MRAM based memory chips [126]. However,

many applications where computations can be non-von Neumann in nature, particularly neuromorphic computing, have leveraged this same characteristic of spintronics to obtain better performance than traditional CMOS-based methods [80]. The absence of stochasticity in CMOS memory/logic necessitates the use of pseudo-random number generators to mimic probabilistic behavior.

In our work, we propose using an MTJ to realize an Ising spin unit since it has 2 stable states, just as is required of an Ising unit. Other non-volatile devices such as RRAMs and PCMs tend to have several intermediate states [144], and therefore, the MTJ is a better choice. It forms the central component of a basic cell of our MTJ-based Ising-FPGA. We exploit its probabilistic switching characteristics to guide the entire system of spins through the states which reduce the energy of the system ($H(x)$ in eqn. 5.1), with the goal of reaching the ground state. Additionally, when the system gets stuck in a local energy minima, the same characteristic would also be used to get it out of the minima. The system of Ising spin units realized with the MTJs would involve interactions among units through voltages and currents which would depend on the parameters of the encoded NP-hard problem and the present state of the system. Details of the implementation shall be discussed shortly.

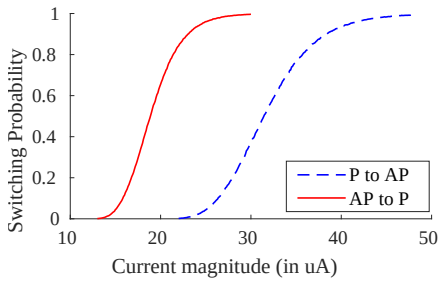


Figure 5.2: Switching probabilities of the MTJ with $2ns$ pulse width. Note that current polarities would be opposite for $P \rightarrow AP$ and $AP \rightarrow P$. Data obtained from MTJ Stochastic LLG simulations [37, 6] in HSPICE, at steps of $0.1\mu A$ with 10000 points per step.

The probabilistic switching of the MTJ is characterized in fig. 5.2. The magnitude of β_i in eqn. 5.4 provides the extent to which x_i can lower the energy of the system, and is thus indicative of the probability with which x_i should change its state (if necessary). For the MTJ-based Ising unit, we can encode the direction and probability with which it should switch in the polarity and magnitude respectively of the switching current provided to it. Although the switching characteristics vary non-linearly with the current as per fig. 5.2, we can perform a linear mapping for simplicity as follows. Considering the gradient in eqn. 5.4, the write current passed through the i^{th} unit may be written as

$$I_i = I_{min} + \frac{\beta_i}{k}\left(I_{max} - I_{min}\right) \tag{5.5}$$

where I_{min} is the minimum current provided to overcome the soft threshold below which the switching probability is negligible, k is a normalizing factor to ensure that I_i is bounded by a maximum current I_{max}. Naturally,

$$k = \max_i(\max|\beta_i|) = \max_i\left(\sum_j^N |J_{ij}| + |h_i|\right) \tag{5.6}$$

which is largest possible influence on any unit. We choose values of I_{min} and I_{max} that correspond to probabilities of roughly 0.1% and 98% respectively for a $2ns$ pulse duration. For $P \rightarrow AP$, $I_{min} = -22\mu A$, $I_{max} = -44\mu A$, and for $AP \rightarrow P$, $I_{min} = 13\mu A$, $I_{max} = 26\mu A$. Note that directly feeding the current obtained from the analog dot product to the MTJ eliminates the use of ADCs.

Once the Ising unit's MTJ is updated probabilistically using the write current

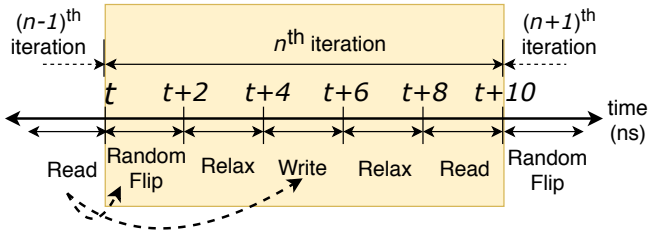


Figure 5.3: Different stages of an iteration in the process of finding the ground state of an Ising model. Each stage is of duration $2ns$, and hence an iteration takes $10ns$. The dashed arrows show where the spin value read is utilized.

in eqn. 5.5, we can allow the magnetization a while to settle, and then read the value stored in the MTJ by passing a small current (say $< 5\mu A$) through it and sensing the potential drop across it [71]. This value read would then be used to update the states of the *other* spins in the next iteration. The effect of random noise in the system can be realized by passing a small current I_{RF} which flips the MTJ with a small probability and, once again, letting it relax.

Fig. 5.3 depicts the timeline of these stages where the Random Flip of a spin unit is done according to its own value read in the *previous* iteration, but before the write stage to avoid another readout. Thus in each cycle/iteration, some of the spin units get updated depending on their interactions with the rest. Ideally, the probability with which this random flip occurs should go down with time in order to maintain an equivalence with the theoretical notion of annealing, which is "cooling the system". Hence the current I_{RF} resulting in random flips must also reduce in magnitude after each cycle. The entire system evolves through several iterations and the Ising energy reduces over time when observed on a large scale, since occasional increases must be expected due to the random flipping.

5.3.3 MTJ-based Ising-FPGA cell

Let us now describe the structure of an Ising spin cell, which is the basic unit of our hardware platform, and show how eqn. 5.5 would be realized. Each Ising cell corresponds to one spin variable and houses the MTJ whose state represents the value of the spin. It is responsible for (a) receiving the states of the other spin units and writing to its MTJ with a certain current, (b) reading the state of its MTJ, and also (c) flipping it randomly.

The coefficients of interactions (J_{ij}) between spin units can be represented by variable resistors, and the summation in eqn. 5.4 can be obtained through an op-amp with $N-1$ inputs. Fig. 5.4 shows the Ising cell in a system with 5 variables. In this figure, we specifically illustrate the Ising cell of variable x_1. It receives binary voltage signals $V_2 \dots V_5 \in \{-V_m, V_m\}$ from the cells of the other variables $x_2 \dots x_5$, where the voltage polarity represents their spin values (V_m for $+1$ and $-V_m$ for -1). These input voltages are modulated by the resistors $R_{12} \dots R_{15}$ and fed to the positive terminal of an op-amp OA1, along with an internal bias voltage V_{h1} through R_{h1}. The output V_o of the op-amp OA1, with feedback resistor R_f, is provided to the MTJ write control circuit shown within the dashed box. It regulates the direction of current I_1 through the MTJ with the help of a pair of switches. These are controlled by the output of comparator OA2 (in open loop configuration) which turns on one and only one of the two switches. The switch controlled by WR is turned on in the Write stage. Voltages V_+ and V_- of opposite polarity are added to V_o to offset it and obtain the minimum current I_{min} for $AP \rightarrow P$ and $P \rightarrow AP$ respectively. Assume without loss of generality that the MTJ can be switched (probabilistically) from

- $AP \rightarrow P$ (that is $-1 \rightarrow 1$) if $V_o > 0$ ($\Rightarrow I_1 > 0$), and
- $P \rightarrow AP$ (that is $1 \rightarrow -1$) if $V_o < 0$ ($\Rightarrow I_1 < 0$)

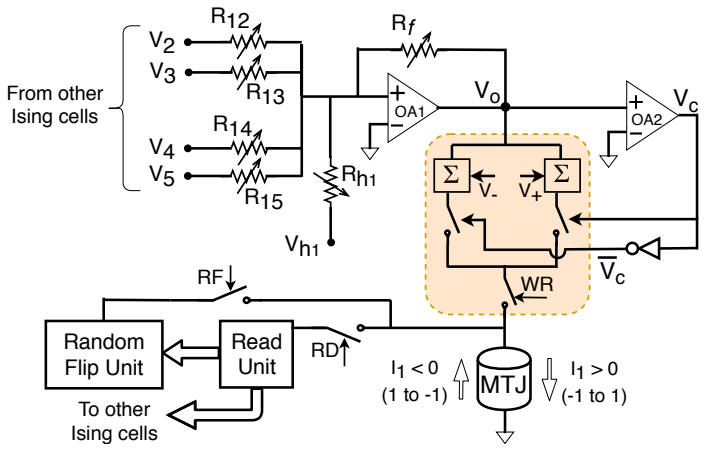


Figure 5.4: The proposed Ising spin cell. Switches WR, RD and RF are turned on in the Write, Read and Random Flip stages respectively.

The output V_o of op-amp OA1 can be expressed as

$$V_o = -R_f \left(\sum_{j=2}^{5} \frac{V_j}{R_{1j}} + \frac{V_{h1}}{R_{h1}} \right) = -R_f \left(\sum_{j=2}^{5} V_j G_{1j} + V_{h1} G_{h1} \right) \tag{5.7}$$

where G denotes the respective conductances. The above relation resembles eqn. 5.4 suggesting that a weighted sum of the outputs from other Ising cells can be easily obtained through an op-amp and resistors. The conductances $G_{ij} \in [G_{min}, G_{max}]$ would be directly proportional to the magnitude of the interaction coefficient, $|J_{ij}|$. If all J_{ij} are normalized such that $|J_{ij}| \leq 1$, then $G_{ij} = |J_{ij}| G_{max}$. To implement bipolar J_{ij}, we can simply add an inverter to each of the $(N-1)$ inputs of x_i's cell to make both V_j and $-V_j$ available, and choose from between the two.

The value of the feedback resistance R_f is dependent on the no. of inputs to the Ising cell and the desired maximum current I_{max}. For an N-variable Ising model, with each Ising cell having $(N-1)$ inputs, we can use the maximum influence in

eqn. 5.6 to calculate the largest possible magnitude of V_o as

$$V_o^{max} = -R_f(k \times (-V_m)G_{max}) = R_f V_m k G_{max} \tag{5.8}$$

Thus, R_f would depend on V_o^{max}, which is in turn decided by I_{max}. In fig. 5.4, parameters $V_+ = 0.227V$, $V_- = -0.172$, $V_o^{max} = 0.184V$, obtained with HSPICE simulations using $V_m = 0.4V, R_P = 5.2k\Omega$, $R_{AP} = 13.7k\Omega$, and values of I_{max} and I_{min} mentioned previously.

The state of the MTJ is sensed by and stored in the Read unit which then provides voltage signals to the other cells (in the next cycle) accordingly. The Random Flip unit sends current I_{RF} to the MTJ to flip it with a small probability, wherein the direction of the current is dependent on the state stored in the Read Unit.

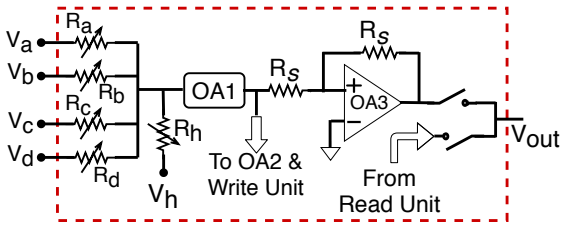
5.3.4 Splitting inputs to multiple cells

Any non-von Neumann hardware platform designed for mapping an Ising-like problem would have a fixed number of inputs per Ising cell, however might it be implemented - spintronics-based [117, 115, 122] or otherwise [140, 85]. Even our Ising-FPGA has a fixed number of inputs per cell. As the problem grows in size, this is going to pose a limitation to the no. of connections made from/to the Ising cells, even if routing may not be an issue.

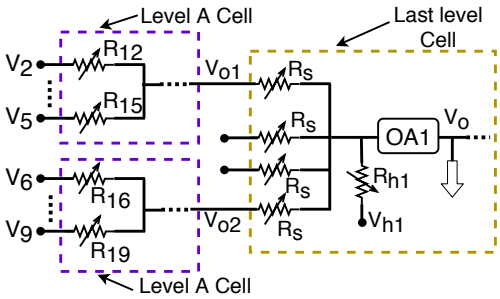
Our approach to dealing with limited fan-in Ising cells is a cascading of several of these cells to accommodate as many inputs as required. The analog nature of the computation in eqn. 5.7 allows for this divide-and-conquer approach with only

a small addition to the basic Ising cell. This is in the form of another op-amp OA3. Fig. 5.5(a) shows the modified Ising cell with $I = 4$ inputs $V_a \ldots V_d$. It can output either from its OA3 or from its Read Unit as required.

Now, considering a fan-in of $I = 4$ per cell, let us show how we can split inputs to a spin variable into multiple Ising cells for an Ising system consisting of 9 variables. The idea is to have several layers/levels (named A,B,...) of the basic Ising cell connected in a tree-like sequence, with outputs from the cells of one level fed into inputs of a cell in the next level until the number of inputs remaining is less than or equal to the fan-in of each cell. The programmable quantities in these cells would be set as required (depending on their level). Fig. 5.5(b) shows how we can split the inputs $V_2 \ldots V_9$ into 2 Ising cells (at level A) which then feeds into the last level cell of variable x_1.



(a) OA3 added to the Ising cell



(b) Splitting 8 inputs into 2 cells

Figure 5.5: (a) The Modified Ising cell. (b) Multi-level Ising cells. Observe that $R_{12} \ldots R_{19}$ are in level A, but R_{h1} is in the last level cell.

Each of the Ising cells would have identical structure and still retain the Write, Read and Random Flip (WR&RF) units, (not shown for simplicity). But there are

certain differences in how the programmable quantities in the cells are set, and how each cell is operated, depending on its level as detailed in table 5.1.

The outputs of the cells shown in fig. 5.5(b) would be

$$V_{o1} = R_s \left(V_2 G_{12} + \ldots V_5 G_{15}\right) \quad \& \quad V_{o2} = R_s \left(V_6 G_{16} + \ldots V_9 G_{19}\right) \tag{5.9}$$

$$V_o = -R_f \left(\frac{V_{o1}}{R_s} + \frac{V_{o2}}{R_s} + \frac{V_{h1}}{R_{h1}}\right) = -R_f \left(\sum_{j=2}^{9} V_j G_{1j} + V_{h1} G_{h1}\right) \tag{5.10}$$

where V_o is the output of last level cell's OA1, and is as desired.

Component/ Quantity	**Level A,B... cells**	**Last level cell**
WR&RF units & MTJ	Disabled (inactive)	Active (MTJ stores x_1)
Bias voltage (V_h)	$0V$	V_{h1} (desired value for x_1)
Output from	OA3	Read Unit
Output sent to	Next level cell	Level A cells of $x_2 \ldots x_9$
Feedback of OA1	Any value (say R_s)	R_f from eqn. 5.8
Input Resistors	For Level A: R_{ij}, Others: R_s	R_s
Bias Resistor (R_h)	Any value (don't care)	R_{h1} (desired bias for x_1)

Table 5.1: Configuration of Ising cells as per their level. Entries of last column specifically for variable x_1.

Ising-FPGA size: Thus, with the proposed approach of splitting the fan-in to several cells, the total no. of levels for every spin variable is $\lceil log_I(N-1) \rceil$, and the no. of cells dedicated to a single variable is $\approx (N-1)/(I-1)$. Hence, the total no. of cells required (with N variables) is *quadratic* in N.

5.4 Architecture of the Ising-FPGA

Field Programmable Gate Arrays (FPGAs) are integrated circuits that offer easy re-programmability, allowing the implementation of any desired logic function [98]. VTR/VPR [84, 7] is an open-source platform for modeling and analyzing FPGA architecture and CAD. The reconfigurable routing topology of the FPGA is a good match for the kind of network connectivity exhibited by an Ising model-based platform such as the one proposed above. The flexibility of connections required by an

Ising solver such as the Ising-FPGA can be fulfilled by the reconfigurability provided by an FPGA-like architecture.

5.4.1 Architecture of an FPGA

Let us first go over the basics of the FPGA architecture before describing how the proposed Ising-FPGA relates to it and how problems can be mapped to the latter. Fig. 5.6 shows the architecture and the traditional interconnect topology of an FPGA. It consists of Configurable Logic Blocks (CLBs) each of which contains a cluster of Basic Logic Elements (BLEs). A BLE is made up of a k-input LUT and provides the LUT's output either directly or through a flip-flop. The interconnects in the FPGA are arranged in several horizontal and vertical channels all around the CLBs, each channel consisting of multiple tracks. The I/O pins of the CLBs are connected to the tracks of the adjacent channels through Connection Boxes (CBs). At the intersection of a vertical and a horizontal channel lies a Switch Box (SB) which is responsible for connecting the tracks of the channels incident on it, thereby facilitating communication between CLBs.

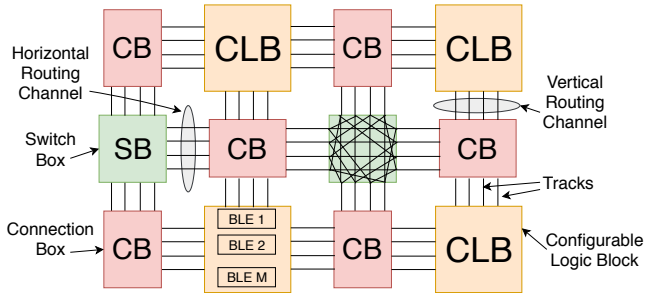


Figure 5.6: The FPGA architecture. CLBs are connected through CBs and SBs.

VTR/VPR [84, 7] is an open-source platform for modeling and analyzing FPGA architecture and CAD. It takes in an architecture file (*.xml*) describing the FPGA, and the circuit's behavioral description in Verilog HDL, and produces an

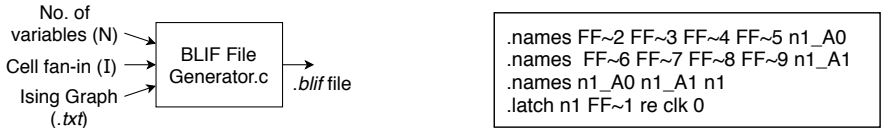
optimized netlist in the Berkeley Logic Interchange Format (BLIF) [1]. In the *.blif* file, a logic gate is declared with a *.names* keyword followed by its inputs and its output. The flip-flops in the BLEs are declared by a *.latch* statement. VPR uses this netlist to pack, place and route the design. It outputs a *.route* file (among others) listing the size of the FPGA, the no. of CLBs in use, and the connections (that is, the nets) from each Source to its Sinks, including the channels that the net passes through.

5.4.2 Reconfigurable Ising model hardware

Let us discuss the analogous of the FPGA's hardware for our Ising-model solver (that is, the Ising-FPGA) and then explain how part of VPR's software flow can be used for configuring the design.

Ising-FPGA: Herein each BLE of an FPGA corresponds to an Ising cell with multiple inputs and one output which can be either the output of OA3 or from the Read Unit, and each CLB contains only one BLE. The size of the LUTs, which in our case would be same as the no. of inputs to the CLB, is set to the no. of inputs to the Ising cell. Thus, for eg. fig. 5.5(b) shows 3 BLEs (or CLBs), each with $I = 4$ inputs. The architecture file (*.xml*) of the FPGA was used to describe certain parameters of the Ising-FPGA.

Recall that each Ising cell in the last level outputs from its Read Unit, whereas cells in other levels output from their OA3. Thus, there is a continuous flow of signal (current) from the last level cell of a spin variable to that of another variable. The equivalent of this for an FPGA is that the BLEs representing last level cells were chosen to output from their flip-flops, while the rest could output straight from



(a) **(b)**

Figure 5.7: (a) The BFG creates the *.blif* file. (b) an excerpt from the *.blif* file responsible for variable x_1 specifying connections in fig. 5.5(b).

their LUTs. The connections between cells is captured by the reprogrammable connectivity of the Ising-FPGA. For our analog design, we can use muxes based on transmission gates (TGs) as switches in the Switch Box (SB), in a way very similar to directional SBs [75]. Thus, a connection between 2 cells has one TG for each SB that it passes through (similar to regular FPGAs).

Using VPR for Ising-FPGA: VPR produces a *.blif* file that describes the netlist of the synthesized network, and uses it to perform place and route of the design. We build a BLIF File Generator (BFG) (fig. 5.7(a)) which takes in the no. of spin variables (N) and the fan-in of each Ising cell (I) as inputs, and creates a *.blif* file by connecting Ising cells in a hierarchical way as demonstrated earlier. Since the *.blif* file should specify only those connections that exist, the BFG also takes in the Ising graph, which lists the pairs of variables (i, j) which have a non-zero interaction ($J_{ij} \neq 0$). VPR uses this *.blif* netlist to pack, place and route the design. It outputs a *.route* file (among many others) that contains the design's routing information.

Fig. 5.7(b) shows a fragment of the *.blif* file generated for the connections[1] pertaining to fig. 5.5(b). Therein, $FF \sim j$ refers to the output from the last level cell of the j^{th} spin variable, $n1\_A0$ and $n1\_A1$ are outputs of the level A cells of x_1, and $n1$ is that of the last level cell of x_1. Hence, the first line describes the I/O

[1]The latch does not indicate a connection from one cell to another and only serves the purpose of marking the end of the combinational circuit.

nets of the level A purple cell of fig. 5.5(b) taking inputs from the last level cells of x_2,x_3,x_4 and x_5, and so on. Since the BFG only connects variable pairs specified in the Ising graph, if $J_{ij} = 0$, then $FF \sim i$ is not an input to any level A cell of x_j (and vice versa) in the *.blif* file.

5.4.3 Signal Degradation and Recovery

The use of TGs for switches in our analog design implies that their finite resistance will result in a potential drop across it, and also bring down the current that was supposed to flow into an Ising cell. We estimate this degradation in every path (from each source cell to its destinations) of the circuit and show how we can recover the original signal.

Fig. 5.8(a) shows the description of one net in the *.route* file provided by VPR. It specifies the location of the source, $n106\_A1$ at $(14, 33)$ in this example, the sections of X- and Y- channels that the net passes through, and the sink/destination ($n106$ at $(13, 34)$). The net is depicted in fig. 5.8(b). It crosses 2 SBs and hence has 2 TGs in its path, and we say it has a path length of 2. We use the information provided in the *.route* file to find the length of the path for each (*src, dest*) pair in the design.

We consider a linear model for the signal degradation, in the sense that the total resistance offered by a path is directly proportional to its length and is independent of the length of other paths (if any) from the same source cell. We use the information provided in the *.route* file to find the length of the path for each $(src, dest)$ pair in the design.

Let us look at the degradation in current before the Level A of Ising cells.

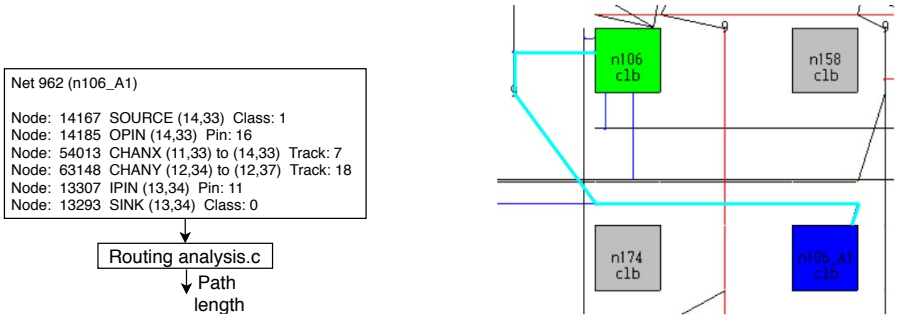


Figure 5.8: Routing of nets. (a) The source, sink and path of a net in textual form in the *.route* file. (b) View of the routing. The source is in dark blue and the sink is in green. The net has been highlighted by us in sky blue.

Fig. 5.9 shows a net from the last level cell of x_j to many level A cells, all with different path lengths. Consider for now the path to level A cell of x_i having length l_{ij}. The current flowing through the input resistance R_{ij} should ideally be V_j/R_{ij}. The presence of TGs, each with resistance R_G, in the path means that this current is now going to be $V_j/(l_{ij}R_G + R_{ij})$. To get back the original current level, we can simply reduce the input resistance R_{ij} by $l_{ij}R_G$ subject to a minimum. The new resistance $\overline{R}_{ij}$ is given as

$$\overline{R}_{ij} = \begin{cases} (R_{ij} - l_{ij}R_G) & \text{if } (R_{ij} - l_{ij}R_G) \geq R_{min}/J_{max} \\ R_{min}/J_{max} & \text{otherwise} \end{cases} \tag{5.11}$$

where $R_{min} = 1/G_{max}$ and $J_{max} \geq 1$ is the largest interaction coefficient for the equivalent of the smallest possible $\overline{R}_{ij}$. Because $\overline{R}_{ij}$ may not still be low enough, we can increase the magnitude of V_j for recovering the desired current. Since different destinations would have different path lengths from the source, they would require to boost V_j by different amounts. Let δ_i^j be the increment in V_j required by the i^{th}

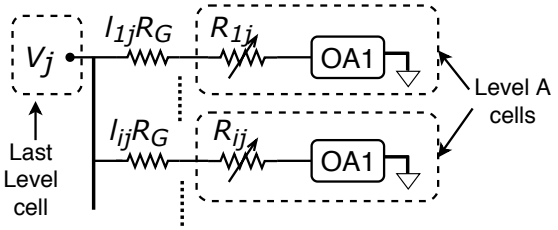


Figure 5.9: Signal degradation model for paths from a last level cell (source) to level A cells (destinations).

destination. Equating the desired and obtained currents,

$$\frac{V_j(1+\delta_i^j)}{\overline{R}_{ij}+l_{ij}R_G} = \frac{V_j}{R_{ij}} \quad \Rightarrow \quad \delta_i^j = \frac{\overline{R}_{ij}+l_{ij}R_G}{R_{ij}} - 1 \tag{5.12}$$

For any source j, the amount of boosting is decided by the destination having the highest value of δ ($\delta_{max}^j = \max_i \delta_i^j$). This boosting can be performed by amplifying the output voltage of the source cell's Read unit through suitable circuits. No extra routing is required for this modification.

Now that V_j has been boosted by δ_{max}^j, the new connection resistances can be obtained yet again by substituting δ_{max}^j in eqn. 5.12. This gives us the final value of the resistors as

$$\overline{R}_{ij} = R_{ij}(1+\delta_{max}^j) - l_{ij}R_G \tag{5.13}$$

For the next level of signal propagation, that is from the output of level A cell to the input of next level's cell, the source connects to only a single destination. Thus, any modifications at the source will depend only on the path for this $(src, dest)$ pair, and can be done by increasing the feedback resistance R_s of the OA1 in the level A cell of the src.

5.5 Ising graphs of NP-hard problems

In this section we describe the combinatorial optimization problems that were mapped to the Ising model and demonstrated using our proposed architecture.

5.5.1 Maximum Cut

Given an undirected graph $G(V, E)$, the Max-cut problem requires partitioning the vertices of G into 2 subsets S and $\overline{S}$ such that the total weight of the edges having one end in S and the other in $\overline{S}$ is maximized. Mathematically, this can be stated as [82]

$$maximize\ \frac{1}{2}\sum_{i,j\in V} W_{ij}(1 - x_i x_j) \tag{5.14}$$

where W_{ij} is the weight of the edge between the i^{th} and j^{th} vertices, and $x_i, x_j \in \{-1, 1\}$ indicate which partition they belong to. Clearly, this objective can be mapped to the Ising Hamiltonian in eqn. 5.1 by choosing $J_{ij} = -W_{ij}/\max_{ij}|W_{ij}|$ (recall that H is minimized, whereas the cut is maximized). This normalizes all the interactions to the range $[-1, 1]$. The bias terms h would be 0 since there is no preferred state ($+1$ and -1 are equivalent).

5.5.2 Travelling Salesman Problem

The TSP is another well-known NP-hard problem which, given N cities and their locations, seeks to find a tour of minimum distance such that each city must be visited exactly once. The Ising formulation of the TSP has a system of N^2 spin variables as shown in fig. 5.10. Each row corresponds to a particular city and each

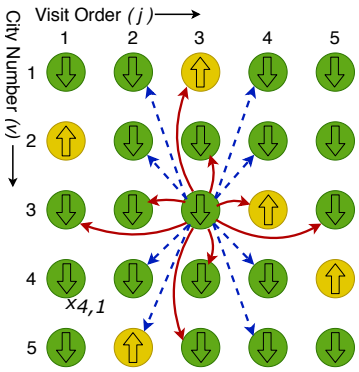


Figure 5.10: Arrangement of Ising spin units for a 5-city TSP. Arrows show interactions of $x_{3,3}$ - solid red ones enforce constraints, whereas dashed blue ones promote progress of tour.

column to a particular visit order. Thus $x_{v,j} = 1$ means that city v is visited in the j^{th} order, whereas $x_{v,j} = 0$ (not -1, note the difference) implies it wasn't visited in the j^{th} order. The Ising Hamiltonian is given as [82]

$$H = \sum_{v=1}^{N} \left(1 - \sum_{j=1}^{N} x_{v,j}\right)^2 + \sum_{j=1}^{N} \left(1 - \sum_{v=1}^{N} x_{v,j}\right)^2 + \lambda \sum_{uvj} W_{uv} x_{u,j} x_{v,j+1} \tag{5.15}$$

Here the first 2 terms ensure that the constraints on the solution to a problem (each city visited exactly once) are satisfied, for which $J_{(v,j)(u,i)} = -1$ whenever $u = v$ or $i = j$. The last term corresponds to the distance travelled in the tour, with W_{uv} being the distance between cities u and v, and λ is a proportionality constant to make sure that the constraints are never violated in favor of a shorter tour, for which the condition $\lambda < 1/\max W(u, v)$ should be satisfied. We have $J_{(v,j)(u,i)} = d_{min}/W_{uv}$, whenever $i = j - 1$ or $j + 1$, where d_{min} is the minimum distance between any pair of cities.

5.6 Simulation Setup and Results

5.6.1 Methodology

Fig. 5.11 depicts the entire flow for simulation and evaluation. First, the nature and parameters of the NP-hard problem are input to the Graph Generator which outputs the interaction matrix J and also the Ising graph. The Ising graph is input to the BLIF File Generator (BFG) which creates the *.blif* file according to the no. of variables (N) and the no. of inputs per Ising cell (I) (sec. 5.4.2). Then, VPR uses the *.blif* and *.xml* files to Place and Route the design. The resultant *.route* file is analysed to obtain the lengths of the path between each pair of connected Ising cells, which is accordingly used to find the degradation in the signals and the modifications necessary in the design (sec. 5.4.3 - Signal Degradation and Recovery - SD&R). This information is passed on to the Stochastic LLG solver along with various other parameters such as the number of iterations to perform, various current values, etc. The LLG simulations of the MTJ were performed using an HSPICE model[2] [37, 6] which was imported into MATLAB for scalability.

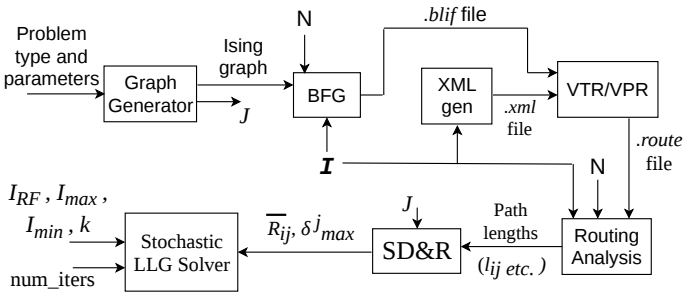


Figure 5.11: Steps performed in the simulations. We start with the Graph Generator and end with the Stochastic LLG simulations.

The current I_{RF} for Random Flipping (sec. 5.3.2) was chosen in a way that it

[2]Device parameters: MTJ cell dimension - $22nm \times 22nm \times 1.5nm$, damping constant $\alpha = 0.01$, simulation time step $\delta_t = 0.01ns$, saturation magnetization $M_s = 800emu/cm^3$

corresponds to roughly 1% switching probability at the beginning of the simulations (at the 1^{st} iteration), and was then reduced linearly to a value that corresponded roughly to 0.1% probability at the end. This is equivalent to the theoretical notion of annealing, which requires "cooling the system".

With regard to accounting for the effects of the hardware, simulations were performed for 3 situations:

- Ideal - Not considering the effects of the underlying hardware, i.e. ignoring signal degradation.
- With Signal Degradation (SD) - Considering the effect of the finite resistances of the paths in the Ising-FPGA, taking $R_G = 3.45k\Omega, R_{min} = 50k\Omega$, but not recovering from the issue.
- Recovery (Rec) - The modifications made in the design to recover the original signals (using $\overline{R}_{ij}, \delta^j_{max}$) with $J_{max} = 10$.

5.6.2 Results

Let us now present the results of the simulations performed for the 2 NP-hard problems. For each of these, we mention the usage of the significant hardware components in the Ising-FPGA. These include

1. the total no. of Ising cells in the Ising-FPGA,
2. the minimum Channel Width Factor (CWF), (the minimum no. of tracks per channel for successful routing),
3. the average length of the paths from the last level cells to the Level A cells (average of all l_{ij} - fig. 5.9) at this CWF.

Max Cut: Table 5.2 specifies the graphs that were used for benchmarking along

with their no. of vertices, the best cut value (obtained using an SDP solver [2]) and the type & range or distribution of edge weights. Table 5.3 lists the aforementioned Ising-FPGA parameters at the specified Ising cell fan-in (I).

Also included is an estimate of the power consumption (in mW) of the system obtained through HSPICE. Fig. 5.12 shows the obtained cut values for the 4 graphs, each normalized by their respective best cut values in table 5.2. Each of the graphs was run 10 times, with 1000 iterations of the Ising simulations per run; all maxcut values are thus average of 10 runs. It is evident that the Ideal maxcut values obtained by simulating the Ising model are very close to the best cut values obtained by heuristics (especially for graphs G1 and G2), thereby revealing the potential of an Ising solver.

Name	Source	Verts	Best Cut	Weight Type & Range
$G1$	G1 from G-set [3]	800	11429	Binary ($\{0, 1\}$)
$G2$	Custom	140	2598.65	Fraction: $U \sim [0, 1]$
$G3$	w01_100.0 from Biq mac [4]	100	645	Integer in $[-10, 10]$
$G4$	ising2.5-300_5555 from [4]	300	8.569×10^6	Int in $[-2, 2] \times 10^5$

Table 5.2: Descriptions of graphs for Maxcut simulations.

Name	G1	G2		G3	G4
I	32	16	32	8	8
No. of cells	2398	1400	840	216	1044
Min. CWF	138	48	48	26	20
Avg. Path Lengths	23.2	8.3	8.3	10.6	5.8
Power	52.37	13.39	14.81	1.02	5.065

Table 5.3: Ising-FPGA hardware usage for Max Cut. Power in mW.

From the data pertaining to fig. 5.12, Signal Degradation (SD) leads to an average relative drop of 1.43% in the MaxCut values. If we define the extent of recovery in the maxcut values as $(Rec - SD)/(Ideal - SD)$, the average recovery across graphs was 78.48%. From table 5.3, we see that a larger fan-in (I) reduces

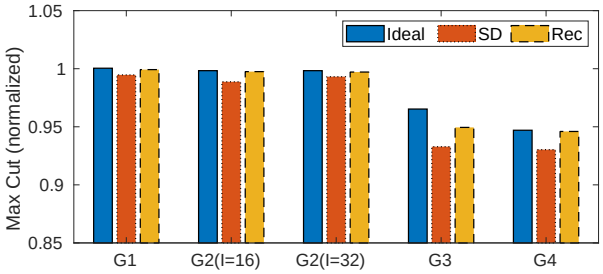


Figure 5.12: Max cut values (normalized) from the Ising simulations for the 4 graphs, with 2 different values of Ising cell fan-in (I) used for G2.

the no. of Ising cells of graph G2 as expected. The minimum CWF and the Average Path Lengths vary in different ways depending on the nature of the graph.

TSP: Three example problems were considered from a dataset [9, 106] - P01, GR17 and FRI26, sets of 15, 17 and 26 cities with optimal tour lengths of 291, 2085 and 937 respectively. Table 5.4 lists the hardware usage on the Ising-FPGA with $I = 16$. Ising simulations were run 20 times (each having 2000 iterations) for each city set. Table 5.5 mentions the results in terms of the no. of runs (out of 20) in which at least 1 "valid" tour was discovered and the average of their Minimum Tour Length (MTL). SD results in an increase in the MTL by an average of 5.86% as compared to Ideal, but, more importantly, it reduces the chances of finding a valid tour. With our recovery strategy, the no. of valid tours is almost as many as those in the Ideal case and the MTL is only 3.49% higher than Ideal on an average.

Name	No.of cells	Min. CWF	Avg.Path lengths	Power
P01	1125	48	8.72	8.17
GR17	1445	48	8.59	12.58
FRI26	5408	60	14.3	45.13

Table 5.4: Ising-FPGA hardware usage for TSP. Power is in mW.

Additionally, fig. 5.13 compares the average no. of valid tours found in each run for the cases Ideal, SD and Rec. Due to SD, this value dropped by an average of 76.83% compared to the Ideal, again indicating reduced chances of finding a valid

tour. We could recover an average of 83.78% of this drop.

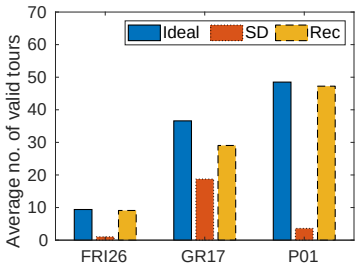


Figure 5.13: Average (over 20 runs) no. of valid tours found in a run.

City set		P01	GR17	FRI26
Valid	Ideal	20	19	16
	SD	9	12	6
	Rec	20	19	19
MTL	Ideal	443	3448	2262
	SD	450	3765	2416
	Rec	453	3689	2290

Table 5.5: Results of Ising simulations for TSP.

5.7 Discussion

Let us now briefly analyze some aspects of our proposed approach and make comparisons with related work.

- **Propagation delay:** Each stage of opamp induces a delay of about $20ps$ (from Cadence Virtuoso simulations). With 3 stages (OA1 & OA3 of level A, and OA1 of last level), the expected propagation delay of Ising spin signals $\pm V_m$ in the write stage is about $0.06ns$. However, this delay could be subsumed within the relax stage just before the write. Further, any minor variations in delay from Ising cell to cell is unlikely to affect the entire system or the final solution, since randomness is an essential part of the Ising computations.
- Resistive RAMs (RRAMs) are a suitable candidate for realizing the variable resistors that capture the interactions between Ising units. These are memristive devices [144, 19] that offer multiple levels of resistance and easy re-programmability.
- Pervaiz et. al. [99] propose the implementation of probabilistic circuits, based on unstable stochastic units called probabilistic bits, on FPGAs. These can be used for Ising and quantum computations. Their entire implementation is on a real

FPGA (and is therefore completely based on digital CMOS logic and memory). On the contrary, our work proposes an FPGA-like architecture based on spintronic and memristive devices so that their inherent randomness and in-memory computing capabilities can be harnessed for realizing an Ising model platform. It is expected to have a much smaller area footprint than a fully digital implementation such as [99]. Since the authors of that work do not report any figures on the area or power consumption of their design, we are unable to make any detailed analysis.

Research on hardware implementations of Ising model typically focuses on the possibility of mapping such models and on obtaining good answers to the associated optimization problem. There is not much emphasis on the characterization of system area/power/performance (yet).

- Process variations in MTJs and RRAMs isn't expected to affect the Ising system to any significant extent, again because such variations add to the randomness in the system which it anyway requires.

5.8 Conclusion

In this chapter, we proposed an Ising model architecture based on MTJs, which can be used to map NP-hard problems and find useful local optimum. We discuss realistic hardware implementations in terms of Ising spin cells and their read/write capabilities, network topology, and re-programmability of interactions among spin units to allow different kinds of NP-hard problems to be encoded. We present Ising-FPGA, a parallel and reconfigurable architecture which can be configured using a standard FPGA Place and Route tool, and discuss ways to incorporate the non-idealities in the hardware into the Ising model.

Chapter 6: Conclusion and Future Work

This thesis demonstrates the potential of MTJs, a spintronic device, in

- accelerating computations through non-von Neumann architectures (such as those based on in-memory processing), which can feasibly be designed/adapted to execute necessary algorithms
- providing a platform for realizing imprecise computing paradigms such as approximate and stochastic computing, and opening doors to optimization for energy-efficiency.

Now we specify, as future work, a possible extension of one of our works and describe one important direction that research with MTJ-based Neural Network hardware can take.

6.1 Ising Graph simplification

Chapter 5 proposed leveraging the Ising model to map and solve NP-hard problems by using MTJs for the hardware realization of Ising spin units. While the architecture proposed by us is non-von Neumann and provides advantages such as parallel computations and reconfigurability of Ising connections, one drawback still remains to be addressed. And that is the explosion in the no. of connections required in a general Ising graph which is quadratic in the no. of Ising units. Although a

quadratic increment is the worst case scenario (since, for eg. an N-city TSP with N^2 Ising units has $\approx 4N^3$ connections), a large-sized problem may find itself unable to be mapped on any Ising-FPGA with reasonable routing capacity. Further, the required no. of Ising cells also grows as $O(N^2)$.

To combat the quadratic growth in hardware resources in the Ising-FPGA, it may be feasible to remove some connections between the Ising units which amounts to doing away with some edges of the Ising graph. That would result in reduced no. of routes between Ising cells and possibly some reduction in the total no. of Ising cells. Recall that the cumulative influence β on any Ising spin unit depends on the interaction strengths J_{ij} between its connections. And it is this β which governs the state update of Ising spins. We plan on investigating into the best ways to simplify the Ising graph that doesn't much impact the β values (and the evolution of the state of the system), while also reducing the hardware usage significantly. The trade-off between resource consumption and solution quality can be analyzed.

6.2 Neuromorphic Computing with Spintronics

This thesis has dealt with the use of MTJs as Stochastic Number Generators in an SC-NN architecture by exploiting its probabilistic switching characteristics, and as analog synapses in crossbar NN architectures. Both of these were in the context of Artificial Neural Networks, where the method of representing and communicating information differs significantly from how the brain does it (hence the term "artificial"). An interesting direction to pursue is looking into the role that MTJs and other spintronic devices can play in the efficient realization of Spiking Neural Networks (SNNs), where data is represented in terms of spike trains. SNNs are a

more accurate model of the brain's working and fit better into what is called Neuromorphic Computing. They can very well be implemented with analog crossbar architectures having spintronic devices as synapses. Also, the switching dynamics of the MTJ and domain wall devices can be leveraged to realize a host of neural activation function. Although there are works (such as [110]) that have investigated into this, there are a plethora of opportunities that emerging non-CMOS devices provide directly and indirectly.

Bibliography

[1] Blif format, 1992.

[2] Computational optimization laboratory, 2002.

[3] The g-set benchmark, 2003.

[4] The biq mac library, 2007.

[5] Training a deep autoencoder or a classifier on mnist digits, 2012.

[6] The llg module, 2016.

[7] Vtr documentation, 2016.

[8] D-wave systems, 2018.

[9] Data for the traveling salesperson problem, 2018.

[10] Deep belief network tutorial, 2018.

[11] Armin Alaghi and John P Hayes. Survey of stochastic computing. *ACM Transactions on Embedded computing systems (TECS)*, 12(2s):92, 2013.

[12] Armin Alaghi, Weikang Qian, and John P Hayes. The promise and challenge of stochastic computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1515–1531, 2017.

[13] Md Z Alom et al. Quadratic unconstrained binary optimization (qubo) on neuromorphic computing system. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3922–3929. IEEE, 2017.

[14] Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J Gross. Vlsi implementation of deep neural network using integral stochastic computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2688–2699, 2017.

[15] SD Bader and SSP Parkin. Spintronics. *Annu. Rev. Condens. Matter Phys.*, 1(1):71–88, 2010.

[16] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[17] Christopher H Bennett, Djaafar Chabi, Theo Cabaret, Bruno Jousselme, Vincent Derycke, Damien Querlioz, and Jacques-Olivier Klein. Supervised learning with organic memristor devices and prospects for neural crossbar arrays. In *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 181–186. IEEE, 2015.

[18] S Bhanja et al. Non-boolean computing with nanomagnets for computer vision applications. *Nature nanotechnology*, 11(2):177, 2016.

[19] M N Bojnordi et al. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE Int. Sym. on High Performance Computer Architecture (HPCA)*, pages 1–13, 2016.

[20] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[21] Bradley D Brown and Howard C Card. Stochastic neural computation. i. computational elements. *IEEE Transactions on computers*, 50(9):891–905, 2001.

[22] Djaafar Chabi, Weisheng Zhao, Damien Querlioz, and Jacques-Olivier Klein. On-chip universal supervised learning methods for neuro-inspired block of memristive nanodevices. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 11(4):34, 2015.

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

[24] Brian Cheung. Convolutional neural networks applied to human face classification. In *2012 11th International Conference on Machine Learning and Applications*, volume 2, pages 580–583. IEEE, 2012.

[25] Leon O Chua and Sung Mo Kang. Memristive devices and systems. *Proceedings of the IEEE*, 64(2):209–223, 1976.

[26] B A Cipra. An introduction to the ising model. *The American Mathematical Monthly*, 94(10):937–959, 1987.

[27] C Cook et al. Gpu based parallel ising computing for combinatorial optimization problems in vlsi physical design. *arXiv preprint:1807.10750*, 2018.

[28] K Corder et al. Solving vertex cover via ising model on a neuromorphic processor. In *2018 IEEE Int. Sym. on Circuits and Systems*, pages 1–5, 2018.

[29] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547 – 553, 2009.

[30] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in NIPS*, pages 3123–3131, 2015.

[31] Lirida Alves de Barros Naviner, Hao Cai, You Wang, Weisheng Zhao, and Arwa Ben Dhia. Stochastic computation with spin torque transfer magnetic tunnel junction. In *New Circuits and Systems Conference (NEWCAS), IEEE 13th International*, pages 1–4. IEEE, 2015.

[32] Erya Deng, Yue Zhang, Jacques-Olivier Klein, Dafiné Ravelsona, Claude Chappert, and Weisheng Zhao. Low power magnetic full-adder based on spin transfer torque mram. *IEEE transactions on magnetics*, 49(9):4982–4987, 2013.

[33] Hoang Anh Du Nguyen, Jintao Yu, Lei Xie, Mottaqiallah Taouil, Said Hamdioui, and Dietmar Fey. Memristive devices for computing: Beyond cmos and beyond von neumann. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–10. IEEE, 2017.

[34] Richard Fackenthal, Makoto Kitagawa, Wataru Otsuka, Kirk Prall, Duane Mills, Keiichi Tsutsui, Jahanshir Javanifard, Kerry Tedrow, Tomohito Tsushima, Yoshiyuki Shibahara, et al. 19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 338–339. IEEE, 2014.

[35] Steve Furber. Large-scale neuromorphic computing systems. *Journal of neural engineering*, 13(5):051001, 2016.

[36] Brian R Gaines. Stochastic computing systems. In *Advances in information systems science*, pages 37–172. Springer, 1969.

[37] S Ganguly et al. Evaluating spintronic devices using the modular approach. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 2:51–60, 2016.

[38] Brian Gardner and André Grüning. Supervised learning in spiking neural networks for precise temporal encoding. *PloS one*, 11(8):e0161335, 2016.

[39] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[40] Tayfun Gokmen and Yurii Vlasov. Acceleration of deep neural network training with resistive cross-point devices: design considerations. *Frontiers in neuroscience*, 10:333, 2016.

[41] R Paul Gorman and Terrence J Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75, 1988.

[42] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[43] H Gyoten et al. Area efficient annealing processor for ising model without random number generator. *IEICE Trans. on Information and Systems*, 101(2), 2018.

[44] H Gyoten et al. Enhancing the solution quality of hardware ising-model solver via parallel tempering. In *2018 IEEE/ACM ICCAD*, pages 1–8, 2018.

[45] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *18th IEEE ETS*, pages 1–6. IEEE, 2013.

[46] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, et al. The ibm blue gene/q compute chip. *Ieee Micro*, 32(2):48–60, 2011.

[47] Raqibul Hasan and Tarek M Taha. Enabling back propagation training of memristor crossbar neuromorphic processors. In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 21–28. IEEE, 2014.

[48] Amr M Hassan, Hai Helen Li, and Yiran Chen. Hardware implementation of echo state networks using memristor double crossbar arrays. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 2171–2177. IEEE, 2017.

[49] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[50] Geoffrey E Hinton. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007.

[51] Geoffrey E Hinton. A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer, 2012.

[52] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[53] William M Holt. 1.1 moore's law: A path going forward. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 8–13. IEEE, 2016.

[54] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Donald Yeung, and Bruce Jacob. Design for reram-based main-memory architectures. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 342–350, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in memory with spin-transfer torque magnetic ram. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483, 2017.

[56] Vahid Jamshidi and Mahdi Fazeli. Pure magnetic logic circuits: A reliability analysis. *IEEE Transactions on Magnetics*, 54(10):1–10, 2018.

[57] Yingyezhe Jin, Yu Liu, and Peng Li. Sso-lsm: A sparse and self-organizing architecture for liquid state machine based neural processors. In *Nanoscale Architectures (NANOARCH), 2016 IEEE/ACM International Symposium on*, pages 55–60. IEEE, 2016.

[58] Andrzej Kasiński and Filip Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal of Applied Mathematics and Computer Science*, 16:101–113, 2006.

[59] Golnar Khodabandehloo, Mitra Mirhassani, and Majid Ahmadi. Analog implementation of a novel resistive-type sigmoidal neuron. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(4):750–754, 2012.

[60] Jongyeon Kim, An Chen, Behtash Behin-Aein, Saurabh Kumar, Jian-Ping Wang, and Chris H Kim. A technology-agnostic mtj spice model with user-defined dimensions for stt-mram scalability studies. In *Custom Integrated Circuits Conference (CICC), 2015 IEEE*, pages 1–4. IEEE, 2015.

[61] Jongyeon Kim, Ayan Paul, Paul A Crowell, Steven J Koester, Sachin S Sapatnekar, Jian-Ping Wang, and Chris H Kim. Spin-based computing: Device concepts, current status, and a case study on a high-performance microprocessor. *Proceedings of the IEEE*, 103(1):106–130, 2015.

[62] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *DAC'16*, page 124. ACM, 2016.

[63] Sungho Kim, Chao Du, Patrick Sheridan, Wen Ma, ShinHyun Choi, and Wei D Lu. Experimental demonstration of a second-order memristor and its ability to biorealistically implement synaptic plasticity. *Nano letters*, 15(3):2203–2211, 2015.

[64] Phil Knag, Wei Lu, and Zhengya Zhang. A native stochastic computing architecture enabled by memristors. *IEEE Transactions on Nanotechnology*, 13(2):283–293, 2014.

[65] Olga Krestinskaya, Khaled Nabil Salama, and Alex Pappachen James. Analog backpropagation learning circuits for memristive crossbar neural networks. In *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pages 1–5. IEEE, 2018.

[66] V Kumar et al. Quantum annealing for combinatorial clustering. *Quantum Information Processing*, 17(2):39, 2018.

[67] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543. ACM, 2008.

[68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[69] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[70] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, Berlin, Heidelberg, 2012.

[71] Hochul Lee, Juan G Alzate, Richard Dorrance, Xue Qing Cai, Dejan Marković, Pedram Khalili Amiri, et al. Design of a fast and low-power sense amplifier and writing circuit for high-speed mram. *IEEE Transactions on Magnetics*, 51(5):1–7, 2015.

[72] Jung Hoon Lee and Konstantin K Likharev. In situ training of cmol crossnets. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 2749–2756. IEEE, 2006.

[73] Jung Hoon Lee and Konstantin K Likharev. Defect-tolerant nanoelectronic pattern classifiers. *International Journal of Circuit Theory and Applications*, 35(3):239–264, 2007.

[74] Robert Legenstein, Christian Naeger, and Wolfgang Maass. What can a neuron learn with spike-timing-dependent plasticity? *Neural computation*, 17(11):2337–2382, 2005.

[75] G Lemieux et al. Directional and single-driver wires in fpga interconnect. In *2004 IEEE Int. Conf. on Field-Programmable Technology*, 2004.

[76] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[77] Z Li and Shufeng Zhang. Magnetization dynamics with a spin-transfer torque. *Physical Review B*, 2003.

[78] M. Lichman. UCI machine learning repository, 2013.

[79] Suhwan Lim, Jong-Ho Bae, Jai-Ho Eum, Sungtae Lee, Chul-Heung Kim, Dongseok Kwon, Byung-Gook Park, and Jong-Ho Lee. Adaptive learning rule for hardware-based deep neural networks using electronic synapse devices. *Neural Computing and Applications*, pages 1–16, 2017.

[80] N Locatelli et al. Spintronic devices as key elements for energy-efficient neuroinspired architectures. In *2015 DATE Conference & Exhibition*. EDA Consortium, 2015.

[81] JW Lu, E Chen, M Kabir, MR Stan, and SA Wolf. Spintronics technology: past, present and future. *International Materials Reviews*, 61(7):456–472, 2016.

[82] A Lucas. Ising formulations of many np problems. *Frontiers in Physics*, 2:5, 2014.

[83] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. Dadiannao: A neural network supercomputer. *IEEE Transactions on Computers*, 66(1):73–88, 2017.

[84] J Luu et al. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Trans. Reconfigurable Tech. and Systems (TRETS)*, 2014.

[85] S Matsumoto et al. Rram/cmos-hybrid architecture of annealing processor for fully connected ising model. In *2018 IEEE Int'l Memory Workshop*, pages 1–4. IEEE, 2018.

[86] Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE, 2011.

[87] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. 74(1):239–255, 2010.

[88] Ankit Mondal and Ankur Srivastava. Power optimizations in mtj-based neural networks through stochastic computing. In *Low Power Electronics and Design (ISLPED), 2017 IEEE/ACM International Symposium on*, pages 1–6. IEEE, 2017.

[89] Ankit Mondal and Ankur Srivastava. In-situ stochastic training of mtj crossbar based neural networks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, page 51. ACM, 2018.

[90] Ankit Mondal and Ankur Srivastava. Energy-efficient design of mtj-based neural networks with stochastic computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(1):1–27, 2019.

[91] Ankit Mondal and Ankur Srivastava. In situ stochastic training of mtj crossbars with machine learning algorithms. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(2):1–29, 2019.

[92] Ankit Mondal and Ankur Srivastava. Spintronics-based reconfigurable ising model architecture. 2020.

[93] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *ICCAD (prijato)*, page 7, 2016.

[94] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in neuroscience*, 7:272, 2014.

[95] F Neumann et al. Combinatorial optimization and computational complexity. In *Bioinspired Computation in Combinatorial Optimization*, pages 9–19. Springer, 2010.

[96] Leibin Ni, Hantao Huang, Zichuan Liu, Rajiv V Joshi, and Hao Yu. Distributed in-memory computing on binary rram crossbar. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):36, 2017.

[97] Dong Pan and Bogdan M Wilamowski. A vlsi implementation of mixed-signal mode bipolar neuron circuitry. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 2, pages 971–976. IEEE, 2003.

[98] H Parvez et al. *Application-specific mesh-based heterogeneous FPGA architectures*. Springer Science & Business Media, 2010.

[99] A Z Pervaiz et al. Weighted p-bits for fpga implementation of probabilistic circuits. *IEEE trans. Neural Networks & Learning Sys.*, 2018.

[100] WJ Poppelbaum. Statistical processors. *Advances in Computers*, 14:187–230, 1976.

[101] WJ Poppelbaum, Apostolos Dollas, JB Glickman, and C O'Toole. Unary processing. In *Advances in computers*, volume 26, pages 47–92. Elsevier, 1987.

[102] Mirko Prezioso, Farnood Merrikh-Bayat, BD Hoskins, GC Adam, Konstantin K Likharev, and Dmitri B Strukov. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550):61–64, 2015.

[103] Damien Querlioz, Olivier Bichler, Philippe Dollfus, and Christian Gamrat. Immunity to device variations in a spiking neural network with memristive nanodevices. *IEEE Transactions on Nanotechnology*, 12(3), 2013.

[104] Nikhil Rangarajan, Arun Parthasarathy, Nickvash Kani, and Shaloo Rakheja. Energy-efficient computing with probabilistic magnetic bits—performance modeling and comparison against probabilistic cmos logic. *IEEE Transactions on Magnetics*, 53(11):1–10, 2017.

[105] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 2016.

[106] G Reinelt. Tsplib: A traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[107] ND Rizzo, D Houssameddine, J Janesky, R Whig, FB Mancoff, ML Schneider, M DeHerrera, JJ Sun, K Nagel, S Deshpande, et al. A fully functional 64 mb ddr3 st-mram built on 90 nm cmos technology. *IEEE Transactions on Magnetics*, 49(7):4441–4446, 2013.

[108] Sylvain Saïghi, Christian G Mayr, Teresa Serrano-Gotarredona, Heidemarie Schmidt, Gwendal Lecerf, Jean Tomas, Julie Grollier, Sören Boyn, Adrien F Vincent, Damien Querlioz, et al. Plasticity in memristive devices for spiking neural networks. 9, 2015.

[109] Sandhya Samarasinghe. *Neural networks for applied sciences and engineering: from fundamentals to complex pattern recognition*. CRC Press, 2016.

[110] Abhronil Sengupta, Aparajita Banerjee, and Kaushik Roy. Hybrid spintronic-cmos spiking neural network with on-chip learning: Devices, circuits, and systems. *Physical Review Applied*, 6, 2016.

[111] Abhronil Sengupta, Maryam Parsa, Bing Han, and Kaushik Roy. Probabilistic deep spiking neural systems enabled by magnetic tunnel junction. *IEEE Transactions on Electron Devices*, 63:2963–70, 2016.

[112] Walter Senn and Stefano Fusi. Convergence of stochastic learning in perceptrons with binary synapses. *Physical Review E*, 71(6):061907, 2005.

[113] Jae-sun Seo, Bernard Brezzo, Yong Liu, Benjamin D Parker, Steven K Esser, Robert K Montoye, Bipin Rajendran, José A Tierno, Leland Chang, Dharmendra S Modha, et al. A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE, 2011.

[114] John M Shalf and Robert Leland. Computing beyond moore's law. *Computer*, 48(12):14–23, 2015.

[115] S Sharmin et al. Magnetoelectric oxide based stochastic spin device towards solving combinatorial optimization problems. *Scientific Reports*, 7(1):11276, 2017.

[116] Ahmad Muqeem Sheri, Aasim Rafique, Witold Pedrycz, and Moongu Jeon. Contrastive divergence for memristor-based restricted boltzmann machine. *Engineering Applications of Artificial Intelligence*, 37:336–342, 2015.

[117] Y Shim et al. Ising computation based combinatorial optimization using spin-hall effect (she) induced stochastic magnetization reversal. *Journal of Applied Physics*, 121(19):193902, 2017.

[118] Daniel Soudry, Dotan Di Castro, Asaf Gal, Avinoam Kolodny, and Shahar Kvatinsky. Memristor-based multilayer neural networks with online gradient descent training. *IEEE transactions on neural networks and learning systems*, 26(10):2408–2421, 2015.

[119] Gopalakrishnan Srinivasan, Abhronil Sengupta, and Kaushik Roy. Magnetic tunnel junction based long-term short-term stochastic synapse for a spiking neural network with on-chip stdp learning. *Scientific reports*, 6:29545, 2016.

[120] Manan Suri, Vivek Parmar, Ashwani Kumar, Damien Querlioz, and Fabien Alibart. Neuromorphic hybrid rram-cmos rbm architecture. In *Non-Volatile Memory Technology Symposium (NVMTS), 2015 15th*, pages 1–6. IEEE, 2015.

[121] Manan Suri, Damien Querlioz, Olivier Bichler, Giorgio Palma, Elisa Vianello, Dominique Vuillaume, Christian Gamrat, and Barbara DeSalvo. Bio-inspired stochastic computing using binary cbram synapses. *IEEE Transactions on Electron Devices*, 60(7):2402–2409, 2013.

[122] B Sutton et al. Intrinsic optimization using stochastic nanomagnets. *Scientific Reports*, 7:44370, 2017.

[123] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. Logic design within memristive memories using memristor-aided logic (magic). *IEEE Transactions on Nanotechnology*, 15(4):635–650, 2016.

[124] Himanshu Thapliyal, Fazel Sharifi, and S Dinesh Kumar. Energy-efficient design of hybrid mtj/cmos and mtj/nanoelectronics circuits. *IEEE Transactions on Magnetics*, 54(7):1–8, 2018.

[125] 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.

[126] L Thomas et al. Basic principles, challenges and opportunities of stt-mram for embedded memory applications. *MSST 2017*, 2017.

[127] Hiroyuki Tomita, Takayuki Nozaki, Takeshi Seki, Toshihiko Nagase, K Nishiyama, E Kitagawa, M Yoshikawa, T Daibou, M Nagamine, T Kishi, et al. High-speed spin-transfer switching in gmr nano-pillars with perpendicular anisotropy. *IEEE Transactions on Magnetics*, 47(6):1599–1602, 2011.

[128] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: Energy-efficient neuromorphic systems using approximate computing. In *ISLPED'14*, pages 27–32. ACM, 2014.

[129] Rangharajan Venkatesan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. Spintastic: spin-based stochastic logic for energy-efficient computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 1575–1578. IEEE, 2015.

[130] Adrien F Vincent, Jérôme Larroque, Nicolas Locatelli, Nesrine Ben Romdhane, Olivier Bichler, Christian Gamrat, Wei Sheng Zhao, Jacques-Olivier Klein, Sylvie Galdin-Retailleau, and Damien Querlioz. Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems. *IEEE transactions on biomedical circuits and systems*, 9(2):166–174, 2015.

[131] KL Wang, JG Alzate, and P Khalili Amiri. Low-power non-volatile spintronic memory: Stt-ram and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003, 2013.

[132] Mengxing Wang, Wenlong Cai, Kaihua Cao, Jiaqi Zhou, Jerzy Wrona, Shouzhong Peng, Huaiwen Yang, Jiaqi Wei, Wang Kang, Youguang Zhang,

et al. Current-induced magnetization switching in atom-thick tungsten engineered perpendicular magnetic tunnel junctions with large tunnel magnetoresistance. *Nature communications*, 9(1):671, 2018.

[133] Qian Wang, Yongtae Kim, and Peng Li. Neuromorphic processors with memristive synapses: Synaptic interface and architectural exploration. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 12(4):35, 2016.

[134] Yandan Wang, Wei Wen, Linghao Song, and Hai Helen Li. Classification accuracy improvement for neuromorphic computing systems with one-level precision synapses. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 776–781. IEEE, 2017.

[135] You Wang, Hao Cai, Lirida AB Naviner, Jacques-Olivier Klein, Jianlei Yang, and Weisheng Zhao. A novel circuit design of true random number generator using magnetic tunnel junction. In *Nanoscale Architectures (NANOARCH), 2016 IEEE/ACM International Symposium on*, pages 123–128. IEEE, 2016.

[136] DC Worledge, G Hu, PL Trouilloud, DW Abraham, S Brown, MC Gaidis, J Nowak, EJ O'Sullivan, RP Robertazzi, JZ Sun, et al. Switching distributions and write reliability of perpendicular spin torque mram. In *Electron Devices Meeting (IEDM), 2010 IEEE International*, 2010.

[137] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2015.

[138] Yan Xu, Xiaoqin Zeng, Lixin Han, and Jing Yang. A supervised multi-spike learning algorithm based on gradient descent for spiking neural networks. *Neural Networks*, 43:99–113, 2013.

[139] Chris Yakopcic, Md Zahangir Alom, and Tarek M Taha. Memristor crossbar deep network implementation based on a convolutional neural network. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 963–970. IEEE, 2016.

[140] M Yamaoka et al. A 20k-spin ising chip to solve combinatorial optimization problems with cmos annealing. *IEEE Journal of Solid-State Circuits*, 51(1), 2016.

[141] Deming Zhang, Lang Zeng, Kaihua Cao, Mengxing Wang, Shouzhong Peng, Yue Zhang, Youguang Zhang, Jacques-Olivier Klein, Yu Wang, and Weisheng Zhao. All spin artificial neural networks based on compound spintronic synapse and neuron. *IEEE transactions on biomedical circuits and systems*, 10(4):828–836, 2016.

[142] Deming Zhang, Lang Zeng, Youguang Zhang, Weisheng Zhao, and Jacques Olivier Klein. Stochastic spintronic device based synapses and spiking neurons for neuromorphic computation. In *Nanoscale Architectures (NANOARCH), 2016 IEEE/ACM International Symposium on*, pages 173–178. IEEE, 2016.

[143] Yaojun Zhang, Xiaobin Wang, Yong Li, Alex K Jones, and Yiran Chen. Asymmetry of mtj switching and its implication to stt-ram designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1313–1318. EDA Consortium, 2012.

[144] Mohammed A Zidan et al. The future of electronics based on memristive systems. *Nature Electronics*, 1(1):22, 2018.