Energy-efficient Design of MTJ-based Neural Networks with Stochastic Computing

ANKIT MONDAL and ANKUR SRIVASTAVA, University of Maryland College Park

Hardware implementations of Artificial Neural Networks (ANNs) using conventional binary arithmetic units are computationally expensive, energy-intensive, and have large area overheads. Stochastic Computing (SC) is an emerging paradigm that replaces these conventional units with simple logic circuits and is particularly suitable for fault-tolerant applications. We propose an energy-efficient use of Magnetic Tunnel Junctions (MTJs), a spintronic device that exhibits probabilistic switching behavior, as Stochastic Number Generators (SNGs), which forms the basis of our NN implementation in the SC domain. Further, the error resilience of target applications of NNs allows approximating the synaptic weights in our MTJ-based NN implementation, in ways brought about by properties of the MTJ-SNG, to achieve energy-efficiency. An algorithm is designed that, given an error tolerance, can perform such approximations in a single-layer NN in an optimal way owing to the convexity of the problem formulation. We then use this algorithm and develop a heuristic approach for approximating multi-layer NNs. Classification problems were evaluated on the optimized NNs and results showed substantial savings in energy for little loss in accuracy.

CCS Concepts: • **Computer systems organization** → **Neural networks**; • **Mathematics of computing** → *Convex optimization*; • **Hardware** → **Spintronics and magnetic technologies**; *Emerging architectures*;

Additional Key Words and Phrases: Magnetic tunnel junctions, energy efficiency, approximate computing, regularization

ACM Reference format:
Ankit Mondal and Ankur Srivastava. 2019. Energy-efficient Design of MTJ-based Neural Networks with Stochastic Computing. *J. Emerg. Technol. Comput. Syst.* 16, 1, Article 7 (October 2019), 27 pages.
https://doi.org/10.1145/3359622

1 INTRODUCTION

The capability of the human brain to learn and solve complex problems has inspired advancements in areas of neuroscience, artificial intelligence, and machine learning. 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 and image classification [50]. However, a typical ANN can have thousands of neurons and synapses, making their hardware implementation both computation- and memory-intensive [41]. This has prompted the

A preliminary version of this work has appeared in the ISLPED 2017 [42].
This work is supported by the National Science Foundation (NSF) under Grant 1642424.
Authors' addresses: A. Mondal and A. Srivastava, 8223 Paint Branch Drive, A V Williams building (Department of Electrical and Computer Engineering), University of Maryland, College Park MD 20742; emails: amondal2@terpmail.umd.edu, ankurs@umd.edu.

1550-4832/2019/10-ART7 $15.00
https://doi.org/10.1145/3359622

development of optimization techniques at different levels of these complex networks to achieve energy efficiency [44, 55].

Approximate Computing is an emerging concept that involves the computation of imprecise results to achieve significant reductions in power consumption [15]. The inherent error-resilience of Recognition, Mining, and Synthesis applications make them a perfect candidate for such trade-off between the quality of results and the energy requirements. A similar paradigm is Stochastic Computing (SC), which concerns the use of low-cost logic gates, instead of binary arithmetic units, for computations [2, 12, 46]. In SC, data, which are interpreted as probabilities and called Stochastic Numbers (SNs), are represented in the form of bit streams of 0s and 1s and generated by circuits called Stochastic Number Generators (SNGs). SC has been shown to be significantly energy-efficient when compared to conventional methods [43]. Traditionally, SNGs are composed of pseudo-random number generators (such as Linear Feedback Shift Registers) and comparators; however, these can account for a significant fraction of the design cost of the complete system in terms of energy and area. For example, the energy can be up to 80% when implemented using CMOS [2, 56]. Thus, designing low-cost SNGs is of prime importance to the overall energy-efficiency of SC-based circuits, and new nanoscale technologies have provided some hope in this regard.

Magnetic Tunnel Junction (MTJ) is one of several emerging spintronic devices [64]. Apart from non-volatility, its high integration density, scalability, and CMOS compatibility make it a suitable candidate for replacing CMOS in future memory devices [27, 28, 57]. The Spin-Transfer Torque RAM, which is based on MTJs, has been explored as a memory device. While a lot of research has focused on reducing its critical switching current density to lower the write energy [20], attempts have been made to exploit the probabilistic switching characteristics of MTJs to use them as SNGs (such as in Reference [8]), which could produce bit streams representing any fraction between 0 and 1. Such applications include the use of MTJs in the implementation of Bayesian inference systems with SC [23] and of synapses in neuromorphic computing systems [63], and for spike generation in spiking neural networks [51].

This article integrates SC based on MTJs into ANNs and explores 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.
- Last, we show how regularization techniques can be incorporated in the NN training process to obtain better results and prove the effectiveness of our algorithm through simulations.

The rest of the article is organized as follows: In Section 2, we provide the necessary background of neural networks and Stochastic Computing and discuss related work. Section 3 describes MTJs and proposes an energy-efficient MTJ-SNG. The algorithm for approximating the weights of NNs is prescribed in Section 4. Section 5 mentions methods for achieving more optimal results from the algorithm. Simulation results are given in Section 6; Section 7 does further analysis of our techniques and Section 8 wraps up our work.

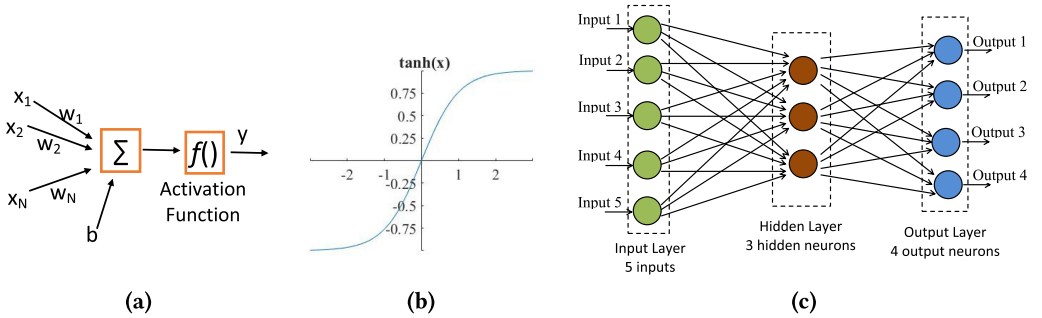


Fig. 1. (a) A neuron. (b) The *tanh* function. (c) Schematic of an MLP with one hidden layer.

2 PRELIMINARIES

2.1 Neural Network Architecture

The fundamental units of an 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 [50]. The output of a neuron, called the *activation level*, can be expressed as

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

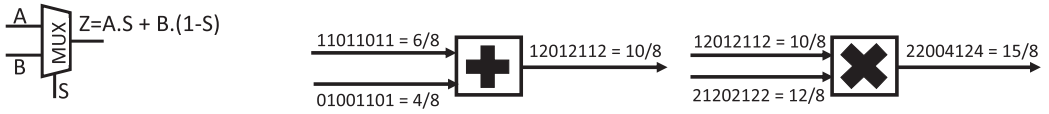
where N is the number of inputs to the neuron, w_i is the synaptic weight of the connection from the ith input x_i, b is a bias, and $f()$ is an activation function (such as $tanh$ or sigmoid). Figure 1(a) depicts the operations performed by a neuron and 1(b), the behavior of the $tanh$ function.

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 feedforward network, called a Perceptron, contains just the input and output layers. More popular and useful are the Multi-layer Perceptrons (MLPs), which have one or more layers of neurons, called hidden layers, between the inputs and the outputs (Figure 1(c)).

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, which are pairs of 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. Weight updates can occur either after each training example is scanned (online learning) or after all of them are scanned (batch training). One single pass/iteration through the entire training dataset is called an *epoch*. This is called Supervised learning, as opposed to Unsupervised Learning, where data is not labelled prior to training and desired outputs are not specified. The most popular technique of training an NN is the *error back-propagation* 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 [50]. This is known as gradient descent or the delta rule and is given as

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \tag{2}$$

where η is known as the learning rate and E is the error function.



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

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

2.2 Stochastic Computing

The concept of Stochastic Computing (SC) and other closely related computational paradigms dates back to the 1960s and '70s [12, 46, 47] and essentially refers to the representation of analog quantities by probabilities of discrete events that 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 $p = \frac{k}{n}$ [2]. 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.

In SC, multiplication is performed by an AND gate in the unipolar format [2]. Thus, given two 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 two 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 Figure 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. [3] introduced 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 $s_1, s_2, \ldots s_m \in [0, 1]$. Each s_i can be represented as stochastic streams and s can be obtained as the bit-wise summation of these m streams, as illustrated by an example in Figure 2(b). For example, 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 two 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 [3]. Multiplication and addition in ISC are performed using binary radix multipliers and adders, respectively. Given two 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. Figure 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.

It is also possible to design good approximations to non-linear functions in ISC using Finite State Machines. Commonly used activation functions in NNs are the hyperbolic tangent (*tanh*), the logistic sigmoid, the ReLU, and the softmax. Brown et al. [5] introduced FSMs based on saturating counters wherein the probabilities of state transition are dependent on the function input and a pair of stochastic control variables. These FSMs can be used to realize any non-linear function; examples for the *tanh*, the linear gain, and the exponential are demonstrated in Reference [5]. The basic FSM designs for *tanh* and exponentiation functions have been extended by Reference [3] for use in the ISC domain. In Reference [36], a ReLU neuron for SC is proposed that adds a history shift register array to the FSM of *tanh* to maintain a non-negative output. An FSM for the logistic sigmoid additionally modifies the output of the states of the FSM to shift the midpoint of the sigmoid's output to 0.5. Finally, it is also possible to perform the softmax function in the ISC domain [19] by making use of several (as many as there are inputs) exponentiation FSMs and an LUT.

2.3 Related Work

Several research efforts have been made both towards the efficient implementation of deep neural networks through approximations (as in References [44], [61], and [55]) and towards the realization of NN hardware with non-conventional methods of computation. In Reference [44], Mrazek et al. provide a methodology for designing a power-efficient NN with a uniform structure using approximate multipliers. The key constraints governing the approximation were determined through an error resilience analysis, with the acceptable error types being specified by a gate level description of the accurate circuit. The algorithm basically performs a design space exploration with the search being guided by an error metric. Zhang et al. [61] propose an approximate computing framework that considers approximating not only the computations but also the memory accesses. They developed an optimization procedure that assesses how critical each neuron is in terms of its impact on output quality and energy consumption and that jointly considers error-tolerance capability and energy consumption. In Reference [55], Venkataramani et al. use back-propagation itself to analyze the neuron criticality, as the process of training is of an error-healing nature. They use precision scaling to design approximate "low-impact" neurons and the weights connected to them, and design a quality-configurable Neuromorphic Processor Engine that provides a programmable hardware for implementing the approximate NNs.

Several works have investigated the use of analog devices for computational purposes. Tarkov [52] proposes using a memristor as a device that stores synaptic weights, thereby obviating the need for a large amount of memory, and develops an algorithm for mapping a weight matrix onto a memristor crossbar. Hu at al. [17] extend this idea by developing a Dot-Product Engine (DPE) for matrix-vector multiplications by taking into account the device and circuit issues. The speed-accuracy product of the DPE was found to be significantly higher than that of a custom digital ASIC. Venkatesan et al. [56] proposed a spintronic-based Stochastic Logic that used the random switching characteristics of a nanomagnet to generate random numbers and MTJs to store them in binary form. MTJs themselves have been proposed as true random number generators by Wang et al. [58], wherein the feasibility of the circuit design is verified by using 28*nm* ultra thin body and buried oxide fully depleted silicon-on-insulator technology. Nickvash et al. [26] have extensively studied the switching characteristics of nanomagnets and developed analytical PDFs for small and large values of current. This is then extended to complex nanomagnet circuits (all-spin logic) and the effect of device-level variations on circuit performance is characterized. In Reference [49], the stochastic switching behavior of a giant spin-Hall device was analyzed and taken advantage of to design low-power probabilistic logic gates, along the lines of an energy-accuracy trade-off. Such a nanomagnet-based implementation was more efficient than probabilistic CMOS logic.

There exists significant literature in the field of Stochastic Computing [6, 29, 34]. Qian et al. [48] synthesize in stochastic logic a reconfigurable architecture that performs processing operations on a datapath. A thorough analysis of the sources of errors that introduce uncertainty in the stochastic operations has been done and a fault tolerance better than conventional (non-SC) architectures has been demonstrated. Also, hardware usage turned out to be significantly less with SC-based implementations for several kinds of applications. In Reference [3], Ardakani et al. design an efficient implementation of an NN in the ISC domain. They achieve significant reduction in power consumption at the same rate of misclassification when compared to CMOS implementation. Kim et al. [29] combine the ideas of SC in DNN and energy-accuracy trade-off by removing near-zero weights during the training phase (and later retraining the network), combining the addition and squashing operations, and incorporating progressive precision in the SC bit streams. Li et al. [37] evaluate Deep Convolutional Neural networks in the Stochastic Computing framework and propose modifications in hardware to account for certain simplifications in the design of SC hardware. An empirical formula for the number of states in an FSM-based *tanh* is developed, and the order of pooling and activation function are reversed, since the activation is more suited to be applied on the higher-precision inner product. These techniques together bring down the network error from 27.83% (for the non-optimized version) to 3.48%. Canals et al. [6] introduce the concept of Extended Stochastic Logic (ESL) to overcome the limitation in the range of numbers represented by conventional SC. Any real number can be encoded in the form of a ratio of two numbers in the standard range. The authors design blocks for performing basic arithmetic functions such as addition, subtraction, multiplication, and division and also specialized ones such as a bipolar divider and hyperbolic tangent. These are then employed in ANNs to solve a 2-D classification problem and a regression, both of which demonstrate close-to-ideal accuracy and noise immunity. The authors of Reference [40] apply SC for the inference in Deep Belief Networks. A reconfigurable activation unit is designed to realize different functions and Stochastic Number generators are shared among neurons in the same layer. Significant benefits are obtained in terms of area, power, and energy consumption as compared to 32-bit floating and 8-bit fixed-point implementations. The work in Reference [35] proposes a hardware-oriented approximate activation function (specifically, a sigmoid) and a new hybrid stochastic multiplier composed of OR gates followed by a binary parallel counter. The presence of OR gates for partial summation reduces the size of the parallel counter. This new architecture saves area, power, and energy by impressive factors.

3 MTJ-BASED STOCHASTIC COMPUTING

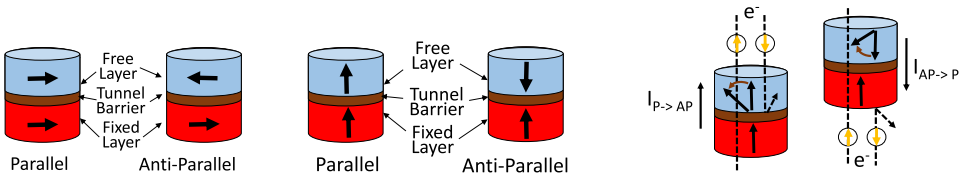
In this section, we shall describe the characteristics of an MTJ with regard to its probabilistic switching and exploit the properties of Stochastic Numbers to design a low-energy optimized MTJ-based SNG and compare it to its non-optimized version. This MTJ-SNG would be the underlying source of approximations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

3.1 Characteristics of Magnetic Tunnel Junctions

MTJ is the most popular spintronic device being considered for NVM technologies [57]. It consists primarily of three layers—two ferromagnetic layers made of CoFeB, and an oxide (typically MgO) layer sandwiched between them acting like a tunnel barrier. An MTJ can exist in one of two stable states, depending on the relative magnetizations of its free and fixed layers—Parallel (P, logic 0) or Anti-Parallel (AP, logic 1). Based on the magnetic anisotropy, MTJs can be broadly classified into two categories [21, 57]:

(1) In-plane MTJ, where the magnetic orientations are in the plane of the tunnel barrier, and
(2) Perpendicular MTJ, where the orientations are perpendicular to that plane.

Figures 3(a) and 3(b) depict these, along with the P and AP configurations.



(a) In-Plane MTJ **(b)** Perpendicular MTJ **(c)** STT-assisted switching in MTJ

Fig. 3. (a) and (b) Schematics of Magnetic Tunnel Junctions with two different magnetic anisotropies, (c) Spin-Torque Transfer switching from P→AP (left) and AP→P (right). Dashed lines show the path of oppositely spin-polarized electrons.

It is possible to flip the magnetization of the free layer of an MTJ by passing spin-polarized current of appropriate polarity. This mechanism, known as Spin-Torque Transfer [28, 64], is illustrated in Figure 3(c). Depending on the magnitude I of the current and the critical current I_{c0} [60], given as[1]

$$I_{c0} = \frac{\alpha\gamma e\mu_0 M_s H_K V}{\mu_B \theta}, \tag{3}$$

the switching behavior of MTJs can be classified into three types: Precessional ($I > I_{c0}$), Dynamic Reversal ($0.8I_{c0} < I < I_{c0}$), and Thermal Activation ($I < 0.8I_{c0}$). The time required for switching is a function of the current I. Further, the process is a stochastic one, which means that current of a given magnitude and pulse width has only a given probability of successfully changing the state of the MTJ. This stochasticity is due to thermal fluctuations in the initial magnetization angle and is an intrinsic property of the STT switching [38]. Since we desire a high-speed SNG, we operate in the Precessional mode, where high currents lead to switching times of the order of a few *ns* [9]. Here the switching probability is expressed as

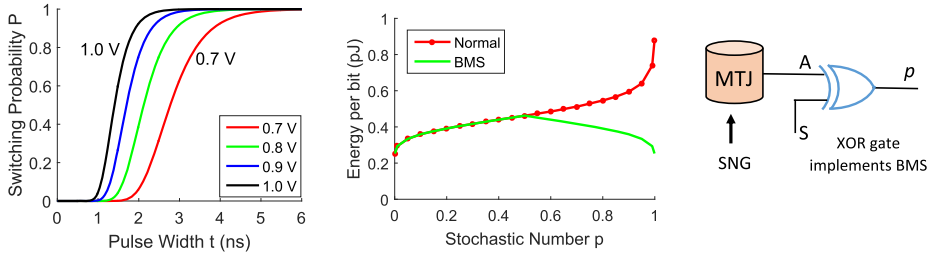
$$\begin{aligned} P(a,t) &= exp(-4f(a)\Delta exp(-2t/T)), \quad with \\ a &= \frac{I}{I_{c0}} \quad and \quad f(a) = \left(\frac{2a}{a-1}\right)^{\left(\frac{-2}{a+1}\right)}, \end{aligned} \tag{4}$$

where t is the pulse width, $\Delta = \frac{H_K M_s V}{2k_B T}$ is the thermal stability, and T is the mean switching time (which is also dependent on a) [54]. The probabilities for AP→P switching, for different voltage bias, are shown in Figure 4(a). The other switching direction exhibits very similar behavior.

The spin transfer efficiency (θ) of an MTJ is different for the two switching directions, with $\theta^{P\to AP}$ having a smaller value than $\theta^{AP\to P}$ [62]. This makes $I_{c0}^{P\to AP} > I_{c0}^{AP\to P}$, which means that the same magnitude and duration of current will correspond to different switching probabilities for the two switching directions. We have simulated the behavior of an MTJ with perpendicular magnetic anisotropy using an MTJ HSPICE Model[2] [27], as it is more scalable and has a lower switching current than its in-plane counterpart. The values of I_{c0} obtained were $64.5\mu A$ for P→AP switching and $21.2\mu A$ for AP→P switching and of Δ was 47.5.

[1]In this formula, and the ones that follow, H_K is the effective anisotropy field, M_s is the saturation magnetization, α is the magnetic damping constant, γ is the gyromagnetic ratio, μ_B is the Bohr magneton, μ_0 is the permeability of free space, V is the volume of the free layer, k_B is the Boltzmann constant, T is the temperature, θ is the spin transfer efficiency, and t_F is the thickness of the free layer.

[2]The parameters used were: cell dimension $35nm \times 35nm$, $t_F = 1.4nm$, $M_s = 1029emu/cm^3$, α (damping constant) = $1.4X10^{-2}$, RA product = $6\Omega\mu m^2$, $T = 300K$. These values are representative of the materials used in the MTJ, which is CoFeB for the ferromagnetic layers and MgO for the tunnel barrier.



(a) Probability v/s pulse width for AP→P

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

(c) Circuit of the BMS

Fig. 4. (a) MTJ stochastic switching characteristics. (b) Variation of energy with value of SN p with and without BMS (green undotted and red dotted lines, respectively). (c) The BMS.

Let us now analyze theoretically the switching time and energy consumption of the MTJ. Given a pulse of width T_p, the expected time at which switching takes place (given it does) is expressed as

$$t_{sw} = \int_0^{T_p} \tau \frac{dP}{dt} d\tau, \tag{5}$$

where the derivative of P is the switching probability density function. Let I_{AP} and I_P denote the currents in the AP and P state, respectively. The expected energy consumed in such a scenario, for AP→P switching, is

$$E_{sw}^{AP\to P} = V\left(I_{AP} t_{sw} + I_P(T_p - t_{sw})\right), \tag{6}$$

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

$$E_{nsw}^{AP\to P} = V I_{AP} T_p. \tag{7}$$

Thus, the expected energy consumed is therefore given as

$$\langle E\rangle^{AP\to P} = P(T_p) E_{sw}^{AP\to P} + (1 - P(T_p)) E_{nsw}^{AP\to P}. \tag{8}$$

Expressions are similar for the P→AP switching.

3.2 MTJ as a Stochastic Number Generator

An MTJ can be used as an SNG by exploiting the probabilistic nature of its switching. 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 not 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 quantify the characteristics of the MTJ in terms of its switching probabilities, average switching times, and expected energy consumption using Equations (4) through (8), with current values (I_P and I_{AP}) taken from the HSPICE model. We obtained that switching from AP→P with 99.9% probability at a voltage bias of $0.9V$ requires a pulse of duration $3.87ns$ and has an expected energy

Table 1. Pulse Width for 50% Switching Probability, Expected Energy Consumption, and the Energy-delay Product (EDP) for $P \rightarrow AP$ Switching with Different Values of Bias Voltages

V_{bias}	$T_p\|_{P=0.5}$ (ns)	$\langle E \rangle$ (pJ)	$T_p\langle E \rangle (\times 10^{-21} Js)$
$0.7V$	3.97	0.374	1.485
$0.8V$	3.22	0.401	1.291
$0.9V$	2.73	0.431	1.177
$1.0V$	2.34	0.463	1.083

consumption of $0.51pJ$. However, for the P→AP direction, the same quantities are higher at $6.39ns$ and $0.81pJ$, respectively (observation of similar nature made in some other works as well, such as Reference [25]). 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.

To decide the bias voltage for the P→AP switch, we assess the energy and delay characteristics of four different voltages with reasonable switching times and use the energy-delay product (EDP) as the metric for comparison. Table 1 specifies, for different bias voltages, the required pulse width for a switching probability of 50%, the expected energy consumption $\langle E \rangle$, and the product of these two quantities, which is equivalent to the EDP. Lower voltages consumed less energy, but had higher pulse width. As per our metric, we choose $1.0V$ as the bias voltage for P→AP switch, since it has the smallest EDP of $1.083 \times 10^{-21} Js$. The red dotted line in Figure 4(b) plots the relation between energy and switching probability at this bias voltage.

Resetting the MTJ to '0' requires a pulse width of $3.87ns$, and switching to '1' with 99.9% probability requires $5.46ns$. Reading the value stored in the MTJ using a sense amplifier can be done with a current of about $1\mu A$ for $2ns$ [33]. Thus, the total time necessary for generating one bit of the SN is (a maximum of) $11.33ns$.

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, generating an SN with value p using the MTJ requires that it switch with the same probability from P→AP. If p is closer to 1 than to 0, then 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 prevent this characteristic from making the SNG energy-intensive, 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). Now all we would need to do is to 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 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 Figure 4(c). The 'S' input can be derived from the most significant bit of the binary number that is being converted to a stochastic number [2]. As an example, if $p = 0.3$, the MTJ-SNG will generate p itself and S will be 0 to output $A = 0.3$. However, 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 Figure 4(b) as a function of p. The symmetry of the plot comes from generating the smaller of p and $1 - p$. Table 2 compares the two 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

Table 2. 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

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 does not scale with the energy, as the write latency also reduces.

3.3.1 Randomness of the MTJ's Behavior. Several applications in cryptography require random numbers of very high quality to keep systems secure. Fukushima et al. [10] experimentally demonstrate the generation of true random numbers using MTJs. Two MTJs meant to generate the number 0.5 had a mean error of only 1.3% and standard deviation of 0.4%. The random bitstreams generated by the two MTJs were found to be independent and had a very low correlation value of $\pm 1 \times 10^{-4}$. Additionally, the MTJ-based random number generator designed in Reference [58] passed 12 out of the 15 NIST SP-800 statistical tests. And the work in Reference [22] tests the suitability of using MTJs for generating Stochastic Numbers and finds low levels of self- and cross-correlation among the generated bitstreams.

We performed LLG simulations (in MATLAB) of MTJ switching to test the quality of the bits using the NIST Statistical Test Suite [1], which checks for randomness. A total of 2^{20} bits were generated (which is the minimum size required by the suite), out of which 49.91% were 1. The bitstream passed all the 15 tests. The Stochastic Computing Correlation (SCC) measurement adopted in SPINBIS [22] yields SCC values within ±0.01 for our bitstreams, which is very close to the ideal value of 0.

3.3.2 Comparison with CMOS-based SNG. The authors in Reference [56] report that a spintronic-based SNG built with the MTJ can be seven times more power-efficient than a CMOS-SNG. Knag et al. [31] synthesize a 100MHz SNG with a 32-bit LFSR and a comparator in 65nm technology, which has a power consumption of 80.2μW. 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. Further, the switching energy of an MTJ depends heavily on the device dimensions that dictate the thermal stability Δ (which is proportional to the volume). A high value of Δ indicates good non-volatility on one hand, but implies large switching current on the other.

4 ENERGY-EFFICIENT MTJ-BASED NN IMPLEMENTATION

Stochastic circuits have gained popularity in low-cost implementation of NNs [3] [29]. 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

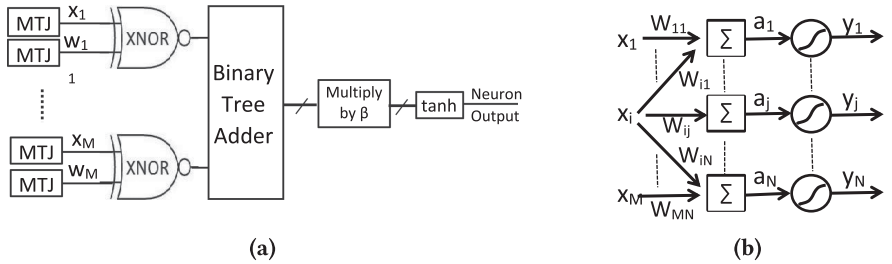


Fig. 5. (a) Neuron implementation in ISC. The outputs of the binary tree adder and multiplier consist of multiple bit-streams. (b) Schematic of 1-layer NN.

dataset and an error tolerance adjust the weights in the best possible way in the solution space, while remaining within the error constraint at all times.

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 2.2). We know that the activation level of a neuron is given as

$$y = f(a) = f\left(\sum_{i=1}^{M} \tilde{w}_i \tilde{x}_i\right), \tag{9}$$

where f is the activation function operating on a, the weighted sum of inputs. Several types of activation functions can be used in an NN. We go for the *tanh* function for the following reasons:

- It is non-linear and has a bipolar output.
- It limits the output to finite range $[-1, 1]$.
- It is continuous and differentiable, enabling us to use the gradient descent method for training.
- It mimics the behavior of biological neurons to a good extent.

In Equation (9), 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. However, 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 use it effectively. Since the ISC implementation of the *tanh* function using FSM is in bipolar format, we go for the interval $[-1, 1]$. So the weighted sum would now be written as

$$a = \beta \sum_{i=1}^{M} w_i x_i, \tag{10}$$

where $x_i, w_i \in [-1, 1]\ \forall\ i$ and would be represented as stochastic numbers. Figure 5(a) illustrates the operations of a neuron in the ISC domain, implementing Equations (9) and (10). The addition, multiplication, and neural activation would be achieved as explained in Section 2.2. Several such neurons in parallel would form a layer as in Figure 1(c), 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.

Let us now briefly describe the FSM-based realization of the *tanh* activation in the SC domain [5]. It essentially composed of a number of states $S_0, S_1 \ldots S_{N-1}$ arranged linearly and mimicking

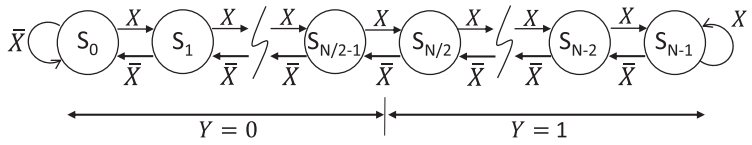


Fig. 6. A Finite State Machine for realizing the non-linear function *tanh* in SC. Each bit of the input bitstream transitions the FSM to the left or right by one state depending on whether it is 0 or 1, respectively, unless the end has been reached.

a saturating counter as shown in Figure 6. Any transition of the FSM can only be to the state in the left or the right, as long as the counter has not saturated. The FSM is initialized at the middle; that is, at state $S_{N/2}$. Upon receiving a '1' from the input bitstream, the machine moves to the state to its right, and left otherwise. It outputs a '0' if the current state is in the left half (that is, states S_0 to $S_{N/2-1}$), and a '1' if it is in the right half. Note, however, that the actual function computed in this way, with N states in the FSM, is $tanh(Nx/2)$. The approximation to the *tanh* obtained using the N-state FSM is good only when N is not very small.

Such a drawback is automatically overcome when implementing the $tanh(x)$ in the ISC domain with a large enough range. As per Reference [3], in the case of m input bitstreams capable of representing numbers in the range $[-m, m]$, the FSM would have m times the number of states as that used for a single bitstream. Further, the maximum number of states that can be transitioned (incremented or decremented) after each time step also goes up to m. The output represented by the FSM states remain the same—0 for the left half, and 1 for the right half. We can express the equivalence between the desired function and the function represented by the stochastic *tanh* as

$$tanh\left(\frac{Nx}{2}\right) \cong 2 \times \mathbb{E}[NStanh(m \times N, X)] - 1. \tag{11}$$

In the above equation, *NStanh* represents the stochastic *tanh* in ISC realized by an FSM with $m \times N$ states, X is the bitstream corresponding to x, and $\mathbb{E}$ is the expectation operator stating the fraction of '1' s in the output bitstream. For example, to obtain the $tanh(x)$ for a number x represented by four bitstreams, the FSM should have $4 \times 2 = 8$ states and is thus capable of providing good enough approximations to the function. As per our notations and Figure 5(a), the neural dot product would be in the range $[-M\beta, M\beta]$ and the FSM for *tanh* would have $2M\beta$ states. Kim et al. [29] provide a lot of insight into how to obtain *tanh* with scaled inputs; such strategy can be put in place as and when necessary.

4.2 Problem Formulation

As can be seen from Figure 4(b), 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.

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

Table 3. 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 rth training input sample	Vector	M
β	The scaling factor for W	Scalar	1
a^r	The rth weighted sums (output layer)	Vector	N
y^r	The rth activation levels (output layer)	Vector	N

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.

Figure 5(b) illustrates a 1-layer network having M inputs and N outputs and Table 3 lists the notations. In addition, the presence of ˆ(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.[3] 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 should not 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 jth neuron for the rth 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 [4]. This is also the reason why we impose a constraint on the range of the weights in line 4 (minimum of distance from 1 and −1 is not 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 that can now be used to check the accuracy of the NN. The optimization problem stated above is

[3]For solving the optimization problems, we use CVX, a package for specifying and solving convex programs [14]. 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($M, N, \hat{W}, \hat{x}, \hat{y}, \beta, \phi$)
(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: **minimize** $\underset{W}{} W_{sod} = \sum_{i=1}^{M}\sum_{j=1}^{N} W'_{ij}$
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

$$W'_{ij} = \begin{cases} 1 + W_{ij} & if\ \hat{W}_{ij} \leq 0, \quad (12) \\ 1 - W_{ij} & otherwise \quad (13) \end{cases}$$

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**

convex, because the objective function and the inequality constraints are convex and the equality constraints are affine [4].

4.4 2-layer NN

A similar formulation could have been made for NNs containing more than one 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 restricting 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 two problems successively—first for the hidden layer and then for the output layer, with error thresholds ϕ_Z and ϕ_W, respectively. Notations used for the 2-layer network, for terms that do not appear in a 1-layer network, are described in Table 4.

4.4.1 Estimation of Maximum Tolerable ϕ_Z. In a 2-layer NN, 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. We know that the weighted sum input to the jth neuron of the output layer is

$$a_j = \sum_{l=1}^{L} W_{lj} h_l. \quad (14)$$

A constraint on the output layer neuron outputs for all of the D inputs is written as

$$|\Delta y_j^r| = |y_j^r - \hat{y}_j^r| \leq \phi_W \quad \forall j = 1 \ldots N, r = 1 \ldots D. \quad (15)$$

Table 4. Notations for Problem Formulation of 2-layer NN

Name	Meaning	Type	Dimension
W	The output layer weights	Matrix	$L \times N$
Z	The hidden layer weights	Matrix	$M \times L$
β_W, β_Z	The scaling factors of W and Z	Scalar	1
b^r	The rth weighted sums of hidden neurons	Vector	L
h^r	The rth hidden neuron outputs	Vector	L

We use a first-order approximation (from Taylor series expansion) to the errors in the weighted sums and write Equation (15) as

$$|a_j^r - \hat{a}_j^r| \leq \frac{\phi_W}{f'(\hat{a}_j^r)} \quad \forall j = 1 \ldots N, r = 1 \ldots D, \tag{16}$$

where f' is the first derivative of *tanh*. Because *tanh* is a monotonically increasing function, f' is always positive. To establish a lower bound, we need to consider the strictest of all constraints, which takes us to

$$|a_j - \hat{a}_j| \leq \min_r \left(\frac{\phi_W}{f'(\hat{a}_j^r)} \right) = \lambda_j(say) \quad \forall j = 1 \ldots N. \tag{17}$$

Because we are interested in deviations in hidden neuron outputs, using Equation (14) and then writing $(h_l - \hat{h}_l) = \epsilon_l$, we obtain

$$\left| \sum_{l=1}^{L} W_{lj}(h_l - \hat{h}_l) \right| = \left| \sum_{l=1}^{L} W_{lj}\epsilon_l \right| \leq \lambda_j \quad \forall j = 1 \ldots N. \tag{18}$$

The LHS of (18) represents a hyperplane (for each j) in the L-dimensional space, with slab constraint (18) having L variables and N equations.

Case 1: $L > N$. The feasible region defined by the inequality in (18) is unbounded. Thus, we can only estimate a lower bound $\bar{\Phi}_Z$ on the maximum error tolerable at the hidden layer. Considering a way of specifying error constraint similar to that of the outer layer, no neuron in the hidden layer should deviate by an amount more than the given ϕ_Z. This can be written as $|\epsilon_l| \leq \phi_Z \; \forall \, l$ or $\|\epsilon\|_\infty \leq \phi_Z$ (restricting the $L\infty$ norm). Let us denote by $\bar{\phi}_j$ the lower bound corresponding to the jth constraint. It can be obtained by finding the point with the smallest $L\infty$ norm that *violates* the jth constraint. That is,

$$\bar{\phi}_j = \min \|\epsilon\|_\infty \quad subject \;\; to \;\; \left| \sum_{l=1}^{L} W_{lj}\epsilon_l \right| \geq \lambda_j. \tag{19}$$

Since the slab constraint should hold for each of the N output neurons (as in (18)), the lower bound can be found as the smallest of all $\bar{\phi}_j$

$$\bar{\Phi}_Z = \min_j \bar{\phi}_j. \tag{20}$$

Because all equations are linear, $\bar{\Phi}_Z$ can be obtained quickly through linear programming.

Case 2: $L \leq N$. The feasible region defined by inequality (18) is bounded. We can find a lower bound using the same argument as above, as well as an upper bound Ψ_Z . The latter will correspond to the largest $L\infty$ norm of the points that satisfy all the constraints. Since maximizing a norm is non-convex, we can maximize the L coordinates one-by-one and then take the largest of these

values. So let $\psi_l = \max \epsilon_l$ with constraint (18) in place. Then upper bound

$$\Psi_Z = \max_l \psi_l = \|\psi\|_\infty. \tag{21}$$

(Note that all the ψ_l will be positive.)

4.4.2 Problem Formulation. Algorithm 1 shows how the weights of the two layers can be optimized independently, with the output from the first being an input to the second. Recall that the error threshold $\bar{\Phi}_Z$ estimated in Equation (20) provides only a lower bound to the maximum tolerable error. Thus, using this estimate may not yield the best approximation possible with the given ϕ_W and it is necessary to solve with higher values of the threshold ϕ_Z and look for better solutions (further approximations). We use a search-based method for this where we start with $\phi_Z = \bar{\phi}_Z$ and keep increasing ϕ_Z in steps as long as it is not large enough to make the optimization problem of the output layer (line 4 of Algorithm 1) infeasible, and then reduce it to within a desired accuracy.

ALGORITHM 1: Problem Formulation for 2-layer NN

1: Obtain $\bar{\phi}_Z$ from ϕ_W using Equation (20) (and Ψ_Z using Equation (21) if relevant)
2: Choose $\phi_Z = \bar{\phi}_Z$ or higher
3: $h = \textsc{OptimWeights}(M, L, \hat{Z}, \hat{x}, \hat{h}, \beta_Z, \phi_Z)$
4: $y = \textsc{OptimWeights}(L, N, \hat{W}, h, \hat{y}, \beta_W, \phi_W)$

We can also generalize the optimization for a network with k layers. Let M_i, N_i denote the number of inputs and outputs of the ith layer, and $\hat{W}_i$ and β_i be the weight matrix and its scaling factor, with $\hat{x}_i$, $\hat{y}_i$ being its inputs and outputs, and ϕ_i being the error tolerance. Algorithm 2 illustrates how the approximation should be performed. The first step is to get the maximum tolerable error for all the hidden layers, starting from the highest one. Then the weights can be optimized layer-by-layer starting from the lowest hidden layer, with approximate outputs of any layer used as inputs to the next layer.

ALGORITHM 2: Problem Formulation for k-layer NN

1: **for** $i = k - 1$ to 1 **do**
2: Obtain $\bar{\phi}_i$ from $\bar{\phi}_{i+1}$ using Equation (20)
3: **end for**
4: **for** $i = 1$ to k **do**
5: Choose $\phi_i = \bar{\phi}_i$ or higher
6: $y_i = \textsc{OptimWeights}(M_i, N_i, \hat{W}_i, \hat{x}_i, \hat{y}_i, \beta_i, \phi_i)$
7: $\hat{x}_{i+1} = y_i$
8: **end for**

5 REGULARIZATION AND CONSTRAINTS FOR CLASSIFICATION PROBLEMS

We now introduce two 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.

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 (E in

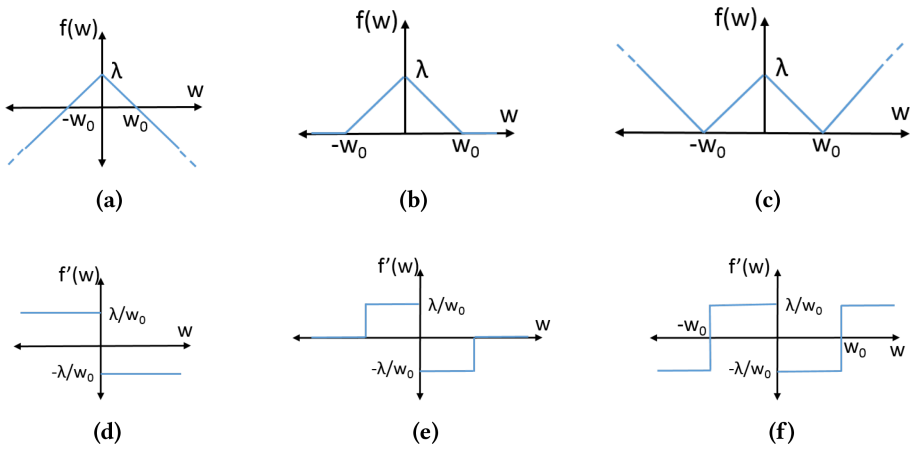


Fig. 7. (a)–(c) Regularization function types 1, 2, and 3, respectively, and (d)–(f) their derivatives.

Equation (2)) 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

$$E = E_I + E_P(W), \tag{22}$$

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

$$\Delta w_i = -\eta \left(\frac{\partial E_I}{\partial w_i} + \frac{\partial E_P}{\partial w_i} \right). \tag{23}$$

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 example, in Reference [59], 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 three 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.

$$f(w) = \lambda \left(1 - \frac{|w|}{w_0} \right). \tag{24}$$

The derivative of this function is given as

$$f'(w) = -\frac{\lambda}{w_0} sign(w). \tag{25}$$

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 Figures 7(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.

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

$$f(w) = \begin{cases} \lambda\left(1 - \frac{|w|}{w_0}\right) & \text{for } -w_0 \leq w \leq w_0, \\ 0 & \text{elsewhere.} \end{cases} \tag{26}$$

Thus, weights that are beyond the range $[-w_0, w_0]$ are not affected by the regularization as depicted in Figures 7(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

$$f(w) = \left|\lambda\left(\frac{|w|}{w_0} - 1\right)\right|. \tag{27}$$

This will try to bring the weights closer to a suitably chosen w_0 as plotted in Figures 7(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 three 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.

5.2 Classification Specific Customization

In Section 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,

$$Class = k = \arg\max_j y_j, \tag{28}$$

with y being the output from the last layer. As long as the kth 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 kth output is only allowed to increase and the rest can only decrease. Mathematically, this means

$$y_k \geq \hat{y}_k \;\; and \;\; y_j \leq \hat{y}_j \; \forall \; j \neq k. \tag{29}$$

This is equivalent to having only a lower limit for the input of the kth neuron and an upper limit for the others,

$$a_k \geq \hat{a}_k \;\; and \;\; a_j \leq \hat{a}_j \; \forall \; j \neq k. \tag{30}$$

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

$$a_k \geq v_k \;\; and \;\; a_j \leq u_j \; \forall \; j \neq k \tag{31}$$

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.

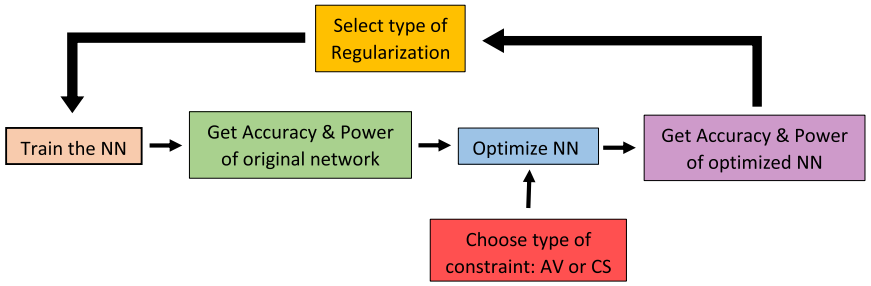


Fig. 8. Flow chart showing the process and network optimization and characterization. We start with training an NN without regularization.

6 EXPERIMENTAL METHODOLOGY AND RESULTS

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 32GB 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 three terms:

- The write energy, which varies with the SN being generated, and which is obtained from the data corresponding to the green plot in Figure 4(b);
- The read energy; and
- The expected energy required to reset, which again depends on the generated SN. The larger the SN, the higher its chances of requiring a reset (although, recall that using BMS means we reset at most half of the times).

Note that the write, read, and reset energy values of the BMS were estimated jointly using the characteristics Equations (4) through (8) and the MTJ HSPICE model [27]. And those of the FSM-based *tanh* have been obtained from Reference [3]. 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 4 for different levels of error tolerance using CVX, a MATLAB-based software tool for solving convex programs [14]. 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 three types of regularization, and each of the four networks were optimized using both types of error constraints—Absolute Value (AV) and Classification Specific (CS). This is illustrated in Figure 8.

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 [32]. A simple 1-layer NN with 784 inputs and 10 outputs was trained—first without and then with the three types of regularization.

Table 5. 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

			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

The 1^{st} column of data corresponds to BMS without any weight approximation.

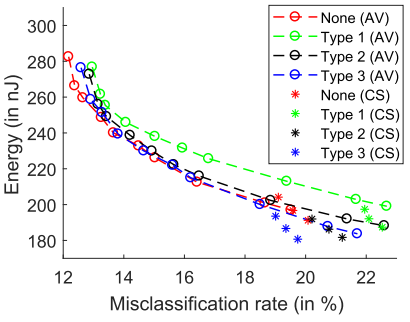


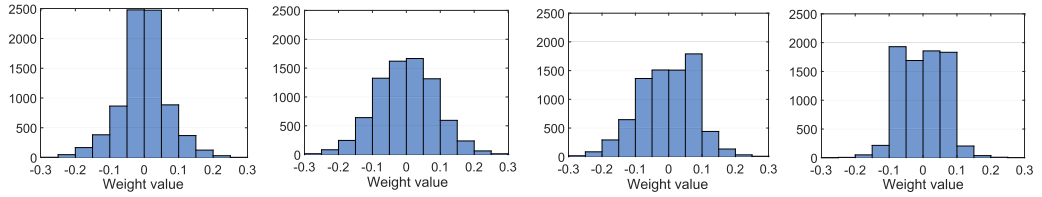
Fig. 9. Energy vs classification error rate curve for the MNIST dataset 1-layer NN. The dots are for the AV constraint, whereas the asterisks are for CS. Optimization was done with 1K training samples.

Table 5 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 Figure 4(b). 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 Figure 9. All accuracy and energy consumption values (here and henceforth) are for 64-bit-long SNs; it varies linearly with the length. The BMS that 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.

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 be noted, however, 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 ($\phi = 0$). However, 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 one side, leading to a larger solution space.

The distribution of weights with and without regularization are shown in Figure 10. Without any penalty on the weight values, the distribution looks roughly Gaussian (Figure 10(a)) with an average magnitude w_0 of about 0.05. With type 1 penalty function, there is a significant drop in the number of weights in the range $[-0.05, 0.05]$ whereas those beyond this range increase. Also, the largest magnitude of weight β increases from 0.312 to 0.386. The type 2 function shows a good concentration of weights just beyond w_0 as expected (because only weights in $[-w_0, w_0]$ incur a penalty). That increases further with the third function, as weights with high magnitude are severely penalized; β drops to 0.270.

Additionally, Table 6 shows the misclassification rates with different lengths of bitstreams of the SNs. A length of ∞ refers to floating point double precision. The degradation in accuracy as it goes down to 16 bits is less than 3% for all but type 1 of regularization. This shows that neural



(a) None **(b)** Type 1 **(c)** Type 2 **(d)** Type 3

Fig. 10. Histograms showing distribution of the weights in the 1-layer NN for different types of regularization.

Table 6. Variation of Classification Error Rate (in %) for MNIST 1-layer NN with Bit-stream Length of the SNs Both Before Weight Approximation (Represented by a '–' Under Threshold) and After (for $\phi = 0$) with the AV Constraint

Regularization	Threshold(ϕ)	Length of Stochastic bitstreams					
		∞	256	128	64	32	16
None	–	12.03	12.10	12.22	11.98	12.15	14.57
	0	12.18	12.20	12.38	12.11	12.21	14.69
Type 1	–	12.71	12.70	12.66	12.83	13.89	16.63
	0	12.96	12.96	12.88	13.07	14.15	16.85
Type 2	–	12.66	12.59	12.82	12.75	13.13	15.03
	0	12.86	12.75	13.01	12.92	13.31	15.19
Type 3	–	12.27	12.25	12.27	12.35	12.59	15.04
	0	12.59	12.58	12.57	12.64	12.91	15.37

computations are quite robust to such quantization; further, the weight optimization process does not take this property away. The latency of inference is $8.21ns$ per bit of the SN (as in Section 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 7. In all energy values, input BMS consumed 8.25 nJ; remaining 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 (first column of Table 7).

The energy-error trade-off with the Absolute Value constraint is depicted in Figure 11. For each kind of regularization (including none), only the points that 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.

Table 7. Results for the MNIST 2-layer Network for Select Values of Error Threshold of the Outer Layer (ϕ_W)

			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

Classification error and energy (in nJ) are for 64-bit-long SNs.

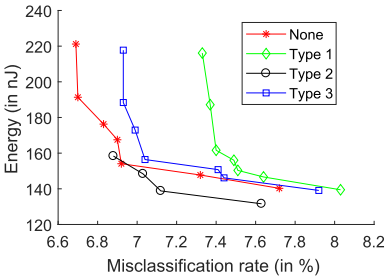
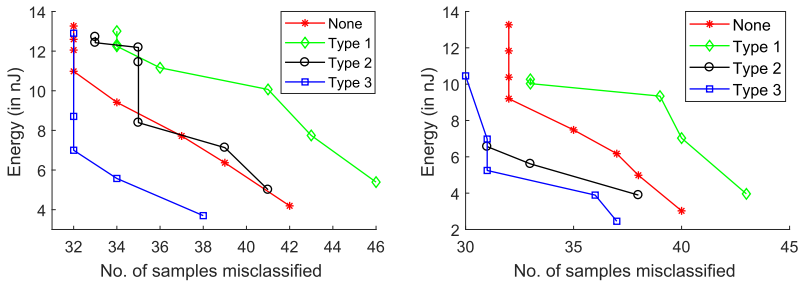


Fig. 11. Plot of classification error rate against energy for 2-layer NN of MNIST dataset with AV constraint.



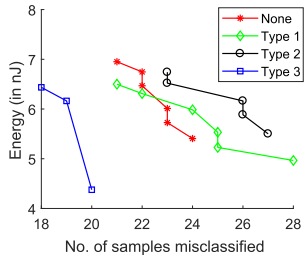
(a) Absolute Value constraint **(b)** Class. Specific constraint

Fig. 12. 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.

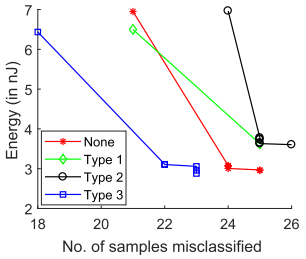
2. Wine Quality: This dataset (as well as the next one) was obtained form the UCI Machine Learning Repository [39]. The goal here is to train a network to estimate the quality of samples of red wine on the basis of results of physiochemical tests [7]. Only a 2-layer NN with 12 input parameters and 20 hidden neurons was trained with 1, 249 samples and tested on 250 samples. The number of misclassified samples before weight approximation and using floating-point precision was 31, 34, 32, and 32 without and with the three types of regularization, respectively. Figure 12 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.

3. SONAR, Rocks vs Mines: This is about distinguishing between metal surfaces and rocks using sonar signals bounced off of them [13]. 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 Figure 13. Before weight approximation, the number of misclassified samples, with floating point double precision, were 20, 21, 24, and 18 for the 1-layer NN and 15, 14, 14, and 13 for the 2-layer NN for None, type 1, 2, and 3, respectively.

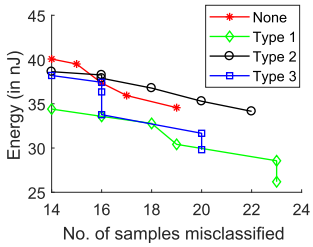
For the 1-layer NN, with both the AV and CS constraints (Figures 13(a) and (b)), type 3 works the best, whereas types 1 and 2 are either similar or worse than None. The CS constraint is better than the AV for all types except type 3, where they are comparable.



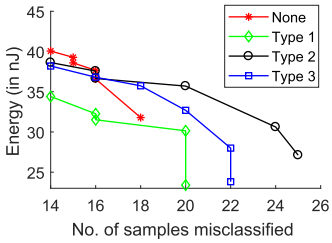
(a) Single-layer NN - AV Constraint



(b) Single-layer NN - CS constraint



(c) Two-layer NN - AV Constraint



(d) Two-layer NN - CS constraint

Fig. 13. Energy vs 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.

Table 8. Execution Time (in seconds) for Training and Optimization of NNs

Dataset		MNIST		Wine	SONAR	
Network		1-layer	2-layer	2-layer	1-layer	2-layer
Training time		338–347	371–398	4	1	2
Optim Time	AV	2,478	1,633	442	4.1	31.7
	CS	2,243	1,479	395	3.6	33.2

MNIST, Wine, and SONAR training typically required 250–300, 80, and 50 epochs, respectively. The values of MNIST training time are in a range due to the slight variations from one type of regularization to another.

In the 2-layer NN, types 1 and 3 outperform None when AV constraint is used (Figure 13(c)), whereas type 2 is a bit worse. With the CS constraint (Figure 13(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.

7 DISCUSSION

In the previous section, we demonstrated the efficacy of our proposed weight-approximation technique by showing substantial reduction in energy consumption for only slight losses in output quality. Also, the use of regularization during training and the Classification Specific error constraint in the optimization is more likely to give better trade-offs than not. Let us now discuss the timing overhead and generality of our approach.

Table 8 lists the time required to train and run the convex optimization program for both types of constraints, which vary widely due to the differences in the network and training set sizes. Training on the MNIST and Wine datasets was stopped using the validation sets. The use of penalty functions did not quite affect the training time, as the matrix-vector multiplications in the

network form the computational bottleneck. For the optimization, although the time requirement is significant for large datasets, the procedure needs to run only once, and it is justified by the obtained energy benefits. Optimization with CS was typically faster by a small margin due to a lesser number of constraints.

Figure 10 revealed that a large number of weights are concentrated around 0. Such distribution is pretty common and has also been observed in many other works for different datasets, such as References [16, 29], indicating a wide relevance of our method.

Several works, such as References [11, 18, 24, 45], and many others, have proposed the use of memristors, another emerging device technology, for the design of neuromorphic systems and random number generators for hardware security applications. Memristors exhibit a stochastic switching behavior, very similar to that of MTJs, which has been leveraged for such purposes. The biased SNG technique proposed by us can also be used in the context of memristors whenever they need to be used as SNGs, and our optimization algorithm would then be applicable for improving energy efficiency. Switching characteristics of memristors as well as MTJs depends a lot on device dimensions and material. From the experimental results reported by these works, memristive switching energy for a 50% probability was calculated to be in a wide range (15 pJ to 0.75 μJ).

There are several works that have looked into the implementation of NN in the SC paradigm. Few of these are theoretical in nature; for instance, the work in Reference [6] introduces the notion of Extended Stochastic Logic (ESL) to go beyond the range of standard SC and cover all real numbers, and design computational blocks for several arithmetic functions. Ardakani et al. [3] develop the concept of Integral SC in the same year for the same purpose and also design necessary logic blocks. Some works optimize or improvise the activation function unit [29, 37, 40] or the addition unit [30, 35, 53]. Yet other works have proposed tweaking standard computations in CNNs [37] or other smart techniques such as early decision termination [29]. The latter work [29] has some similarity with our work in one aspect—it, too, advocates removing near-zero weights, but the intention is to reduce random errors from the XNOR gate (the multiplier).

In our work, we focus only on optimizing the SNG part of the entire SC circuit, leaving the multipliers, adders, and activation units untouched. Our BMS exploits the probabilistic character of the MTJ (something that could exist only in an analog devices) and the resultant energy consumption pattern. And our weight approximation procedure is an entirely software-based optimization approach; that is, it does not ask for any modifications in the SC circuits.

8 CONCLUSION

This article 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 three 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. Last, 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 work around 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.

REFERENCES

[1] National Institute of Standards and Technology. 2017. The NIST Statistical Test Suite. Retrieved from: https://github.com/arcetri/sts.

[2] Armin Alaghi and John P. Hayes. 2013. Survey of stochastic computing. *ACM Trans. Embedd. Comput. Syst.* 12, 2s (2013), 92.

[3] Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. 2016. VLSI implementation of deep neural networks using integral stochastic computing. In *Proceedings of the 9th International Symposium on Turbo Codes and Iterative Information Processing (ISTC'16)*. IEEE, 216–220.

[4] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.

[5] Bradley D. Brown and Howard C. Card. 2001. Stochastic neural computation. I. Computational elements. *IEEE Trans. Comput.* 50, 9 (2001), 891–905.

[6] Vincent Canals, Antoni Morro, Antoni Oliver, Miquel L. Alomar, and Josep L. Rosselló. 2016. A new stochastic computing methodology for efficient neural network implementation. *IEEE Trans. Neural Netw. Learn. Syst.* 27, 3 (2016), 551–564.

[7] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. 2009. Modeling wine preferences by data mining from physicochemical properties. *Dec. Supp. Syst.* 47, 4 (2009), 547–553. DOI: https://doi.org/10.1016/j.dss.2009.05.016.

[8] Lirida Alves de Barros Naviner, Hao Cai, You Wang, Weisheng Zhao, and Arwa Ben Dhia. 2015. Stochastic computation with spin torque transfer magnetic tunnel junction. In *Proceedings of the IEEE 13th International New Circuits and Systems Conference (NEWCAS'15)*. IEEE, 1–4.

[9] Zhitao Diao, Zhanjie Li, Shengyuang Wang, Yunfei Ding, Alex Panchula, Eugene Chen, Lien-Chang Wang, and Yiming Huai. 2007. Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. *J. Phys.: Cond. Matt.* 19, 16 (2007), 165209.

[10] Akio Fukushima, Takayuki Seki, Kay Yakushiji, Hitoshi Kubota, Hiroshi Imamura, Shinji Yuasa, and Koji Ando. 2014. Spin dice: A scalable truly random number generator based on spintronics. *Appl. Phys. Exp.* 7, 8 (2014), 083001.

[11] Siddharth Gaba, Patrick Sheridan, Jiantao Zhou, Shinhyun Choi, and Wei Lu. 2013. Stochastic memristive devices for computing and neuromorphic applications. *Nanoscale* 5, 13 (2013), 5872–5878.

[12] Brian R. Gaines. 1969. Stochastic computing systems. In *Advances in Information Systems Science*. Springer, 37–172.

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

[14] Michael Grant and Stephen Boyd. 2014. CVX: Matlab Software for Disciplined Convex Programming, version 2.1. Retrieved from: http://cvxr.com/cvx.

[15] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 18th IEEE European Test Symposium*. IEEE, 1–6.

[16] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1135–1143.

[17] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R. Stanley Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Proceedings of the 53rd Design Automation Conference (DAC'16)*. IEEE, 1–6.

[18] Miao Hu, Yu Wang, Qinru Qiu, Yiran Chen, and Hai Li. 2014. The stochastic modeling of TiO 2 memristor and its usage in neuromorphic system design. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. IEEE, 831–836.

[19] Ruofei Hu, Binren Tian, Shouyi Yin, and Shaojun Wei. 2018. Optimization of softmax layer in deep neural network using integral stochastic computation. *J. Low Pow. Electron.* 14, 4 (2018), 475–480.

[20] Yiming Huai. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bull.* 18 (2008), 33–40.

[21] S. Ikeda, K. Miura, H. Yamamoto, K. Mizunuma, H. D. Gan, M. Endo, S. L. Kanai, J. Hayakawa, F. Matsukura, and H. Ohno. 2010. A perpendicular-anisotropy CoFeB–MgO magnetic tunnel junction. *Nat. Mat.* 9, 9 (2010), 721.

[22] Xiaotao Jia, Jianlei Yang, Pengcheng Dai, Runze Liu, Yiran Chen, and Weisheng Zhao. 2019. SPINBIS: Spintronics based Bayesian inference system with stochastic computing. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* (2019). DOI: 10.1109/TCAD.2019.2897631

[23] Xiaotao Jia, Jianlei Yang, Zhaohao Wang, Yiran Chen, Hai Helen Li, and Weisheng Zhao. 2018. Spintronics based stochastic computing for efficient Bayesian inference system. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference (ASP-DAC'18)*. IEEE, 580–585.

[24] Hao Jiang, Daniel Belkin, Sergey E. Savel'ev, Siyan Lin, Zhongrui Wang, Yunning Li, Saumil Joshi, Rivu Midya, Can Li, Mingyi Rao, et al. 2017. A novel true random number generator based on a stochastic diffusive memristor. *Nat. Commun.* 8, 1 (2017), 882.

[25] Wang Kang, Tingting Pang, Bi Wu, Weifeng Lv, Youguang Zhang, Guanyu Sun, and Weisheng Zhao. 2016. PDS: Pseudo-differential sensing scheme for STT-MRAM. In *Proceedings of the 53rd Design Automation Conference*. ACM, 120.

[26] Nickvash Kani, Shaloo Rakheja, and Azad Naeemi. 2016. A probability-density function approach to capture the stochastic dynamics of the nanomagnet and impact on circuit performance. *IEEE Trans. Elect. Dev.* 63, 10 (2016), 4119–4126.

[27] Jongyeon Kim, An Chen, Behtash Behin-Aein, Saurabh Kumar, Jian-Ping Wang, and Chris H. Kim. 2015. A technology-agnostic MTJ SPICE model with user-defined dimensions for STT-MRAM scalability studies. In *Proceedings of the Custom Integrated Circuits Conference (CICC'15)*. IEEE, 1–4.

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

[29] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. 2016. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the Design Automation Conference*. ACM, 124.

[30] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. 2015. Approximate de-randomizer for stochastic circuits. In *Proceedings of the International SoC Design Conference (ISOCC'15)*. IEEE, 123–124.

[31] Phil Knag, Wei Lu, and Zhengya Zhang. 2014. A native stochastic computing architecture enabled by memristors. *IEEE Trans. Nanotechnol.* 13, 2 (2014), 283–293.

[32] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. 1998. The MNIST database of handwritten digits. Retrieved from http://yann.lecun.com/exdb/mnist.

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

[34] Bingzhe Li, M. Hassan Najafi, and David J. Lilja. 2019. Low-cost stochastic hybrid multiplier for quantized neural networks. *ACM J. Emerg. Technol. Comput. Syst.* 15, 2 (2019), 18.

[35] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J. Lilja. 2019. Neural network classifiers using a hardware-based approximate activation function with a hybrid stochastic multiplier. *ACM J. Emerg. Technol. Comput. Syst.* 15, 1 (2019), 12.

[36] Ji Li, Zihao Yuan, Zhe Li, Caiwen Ding, Ao Ren, Qinru Qiu, Jeffrey Draper, and Yanzhi Wang. 2017. Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'17)*. IEEE, 1230–1236.

[37] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Yanzhi Wang, and Bo Yuan. 2016. DSCNN: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks. In *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD'16)*. IEEE, 678–681.

[38] Z. Li and Shufeng Zhang. 2003. Magnetization dynamics with a spin-transfer torque. *Phys. Rev. B* (2003).

[39] M. Lichman. 2013. UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml.

[40] Yidong Liu, Yanzhi Wang, Fabrizio Lombardi, and Jie Han. 2018. An energy-efficient stochastic computational deep belief network. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. IEEE, 1175–1178.

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

[42] Ankit Mondal and Ankur Srivastava. 2017. Power optimizations in MTJ-based neural networks through stochastic computing. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED'17)*. IEEE, 1–6.

[43] Bert Moons and Marian Verhelst. 2014. Energy-efficiency and accuracy of stochastic computing circuits in emerging technologies. *IEEE J. Emerg. Select. Top. Circ. Syst.* 4, 4 (2014), 475–486.

[44] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. 2016. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *Proceedings of the International Conference on Computer Aided Design*. 7.

[45] Rawan Naous, Maruan AlShedivat, Emre Neftci, Gert Cauwenberghs, and Khaled Nabil Salama. 2016. Memristor-based neural networks: Synaptic versus neuronal stochasticity. *AIP Adv.* 6, 11 (2016), 111304.
[46] W. J. Poppelbaum. 1976. Statistical processors. *Adv. Comput.* 14 (1976), 187–230.
[47] W. J. Poppelbaum, Apostolos Dollas, J. B. Glickman, and C. O'Toole. 1987. Unary processing. In *Advances in Computers*. Vol. 26. Elsevier, 47–92.
[48] Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. 2011. An architecture for fault-tolerant computation with stochastic logic. *IEEE Trans. Comput.* 60, 1 (2011), 93–105.
[49] Nikhil Rangarajan, Arun Parthasarathy, Nickvash Kani, and Shaloo Rakheja. 2017. Energy-efficient computing with probabilistic magnetic bits—Performance modeling and comparison against probabilistic CMOS logic. *IEEE Trans. Magnet.* 53, 11 (2017), 1–10.
[50] Sandhya Samarasinghe. 2016. *Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*. CRC Press.
[51] Gopalakrishnan Srinivasan, Abhronil Sengupta, and Kaushik Roy. 2017. Magnetic tunnel junction enabled all-spin stochastic spiking neural network. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 530–535.
[52] M. S. Tarkov. 2015. Mapping weight matrix of a neural network's layer onto memristor crossbar. *Optic. Mem. Neural Netw.* 24, 2 (2015), 109–115.
[53] Pai-Shun Ting and John Patrick Hayes. 2014. Stochastic logic realization of matrix operations. In *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD'14)*. IEEE, 356–364.
[54] Hiroyuki Tomita, Takayuki Nozaki, Takeshi Seki, Toshihiko Nagase, K. Nishiyama, E. Kitagawa, M. Yoshikawa, T. Daibou, M. Nagamine, T. Kishi, et al. 2011. High-speed spin-transfer switching in GMR nano-pillars with perpendicular anisotropy. *IEEE Trans. Magnet.* 47, 6 (2011), 1599–1602.
[55] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'14)*. ACM, 27–32.
[56] Rangharajan Venkatesan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. 2015. Spintastic: Spin-based stochastic logic for energy-efficient computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'15)*. IEEE, 1575–1578.
[57] K. L. Wang, J. G. Alzate, and P. Khalili Amiri. 2013. Low-power non-volatile spintronic memory: STT-RAM and beyond. *J. Phys. D: Appl. Phys.* 46, 7 (2013), 074003.
[58] You Wang, Hao Cai, Lirida A. B. Naviner, Jacques-Olivier Klein, Jianlei Yang, and Weisheng Zhao. 2016. A novel circuit design of true random number generator using magnetic tunnel junction. In *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'16)*. IEEE, 123–128.
[59] Yandan Wang, Wei Wen, Linghao Song, and Hai Helen Li. 2017. Classification accuracy improvement for neuromorphic computing systems with one-level precision synapses. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. IEEE, 776–781.
[60] Deming Zhang, Lang Zeng, Youguang Zhang, Weisheng Zhao, and Jacques Olivier Klein. 2016. Stochastic spintronic device based synapses and spiking neurons for neuromorphic computation. In *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'16)*. IEEE, 173–178.
[61] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. 2015. ApproxANN: An approximate computing framework for artificial neural network. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 701–706.
[62] Yaojun Zhang, Xiaobin Wang, Yong Li, Alex K. Jones, and Yiran Chen. 2012. Asymmetry of MTJ switching and its implication to STT-RAM designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 1313–1318.
[63] Yue Zhang, WeiSheng Zhao, Jacques-Olivier Klein, Wang Kang, Damien Querlioz, Claude Chappert, and Dafiné Ravelosona. 2013. Multi-level cell spin transfer torque MRAM based on stochastic switching. In *Proceedings of the 13th IEEE Conference on Nanotechnology (IEEE-NANO'13)*. IEEE, 233–236.
[64] Jian-Gang, Jimmy Zhu, and Chando Park. 2006. Magnetic tunnel junctions. *Mat. Tod.* 9, 11 (2006), 36–45.

Received July 2018; revised June 2019; accepted August 2019