UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Science
Optoelectronics Research Center

Memristor-based Spiking Neural Networks

by

Jinqi Huang
ORCiD: 0000-0003-3913-0067

A thesis for the degree of
Doctor of Philosophy

November 2022

ABSTRACT

Emerging non-volatile memory devices, known as memristors, have demonstrated remarkable perspective in neuromorphic hardware designs, particularly in spiking neural network (SNNs) hardware implementation. Memristor-based SNNs have been applied in solving tasks (e.g. image classification and pattern recognition) traditionally solved by conventional artificial neural networks (ANNs), and more attempts in varying disciplines are still being made to exploit the potential of this new research topic. To apply memristors in neuromorphic applications (strictly defined as applications using SNNs in this thesis), two pathways can be followed. One starts by characterising and controlling memristor devices by utilising hardware infrastructure, which is later mapped with the application's higher-level functions (e.g. matrix multiplications). Another embeds data-driven memristor models in software simulators to emulate the application with parameters extracted from real devices.

This thesis aims to build a cohesive pipeline for bringing memristor-based SNNs to practical use following these two pathways. To achieve this goal, three key designs have been developed. The first one is an FPGA-based digital interface that is part of a memristor characterisation and control system which enables 64-channel parallel read/write operations and high-speed data processing. The control system, developed by the author and two other researchers, not only acts as a testing tool for collecting memristor characteristics but also delivers higher-level functions with memristor arrays in neuromorphic designs. Whilst the thesis focuses on a usage scenario for memristors, this system includes more powerful, versatile testing functionality for other two-terminal emerging memory devices. The FPGA-based interface was developed by the author solely, and it achieves 64-channel level parallelism in the application aspect and complex digital system design and organisation in the engineering aspect. The digital interface is validated by resistor handling and current-voltage sweep experiments.

The second design is the first Python-based algorithm-level simulator, NeuroPack, for memristor-based SNNs with a data-driven memristor model. NeuroPack aims to allow users to quickly validate a neuromorphic concept in the pre-hardware design phase. This tool provides a wide range of optional neuron, plasticity, and device models for users to choose from and answers a fundamental question: is the design functional given the knowledge of the memristor switching dynamics, assuming that the rest of the design is functionally perfect? If yes, the design can move ahead towards the next step. Besides, NeuroPack stores internal variables, including membrane voltages, neuron firing history, and memristor states. With a built-in analysis tool, users can analyse and visualise inference results, observe the evolution of weights and membrane voltages and monitor memristor behaviours. A handwritten digit recognition task in the MNIST dataset showcased how NeuroPack assists users in confirming system validation and exploring sensitivity to critical design choices.

The third design tries to expand the usage of memristor-based SNNs to high-dimensional large-scale applications. To do so, a bespoke simulation framework extended from the second design is developed. The first sentiment analysis task in the IMDB movie reviews dataset is exhibited with this framework. Two paths are taken to train spiking neural networks with memristor models: 1) by converting a pre-trained artificial neural network (ANN) to a memristor-based SNN, or 2) by directly training a memristor-based SNN. These two paths have two application scenarios: offline classification and online training. By converting a pre-trained ANN to a memristor-based SNN and training the mem ristor-based SNN directly, we achieve a classification accuracy of 85.88% and 84.86%, respectively, with the equivalent ANN achieving a baseline training accuracy of 86.02%. From ANNs to SNNs and from non-memristive synapses to data-driven memristive synapses, comparable classification accuracy can be achieved in simulation. In addition, investigations of the neural network sensitivity to global parameters such as spike train length, the read noise, and the weight updating stop conditions have also been given. These investigations further suggest that the simulation framework with statistic memristor models that use experimental data for statistical fitting taking the two paths presented in this chapter can help to exploit the potential of incorporating memristor-based SNNs in text classification tasks.

In summary, with the aid of the designs presented in this thesis, we envisage the two pathways now are complete to achieve neuromorphic applications with memristors, especially in performing text classification tasks. The thesis is concluded with the achieved contributions and future perspectives toward AI hardware.

Contents

List of Figures

List of Tables

Author’s Publication List

- **Jinqi Huang**, Alexantrou Serb, Spyros Stathopoulos, and Themis Prodromakis: **Text Classification in Memristor-based Spiking Neural Networks**. *arXiv e-prints*, 2022, arXiv:2207.13729.
- **Jinqi Huang**, Spyros Stathopoulos, Alexantrou Serb and Themis Prodromakis: **NeuroPack: An Algorithm-level Python-based Simulator for Memristor-empowered Neuro-inspired Computing**. *Frontiers in Nanotechnology*, 2022, doi: 10.3389/fnano.2022.851856.
- **Jinqi Huang**, Spyros Stathopoulos, Alex Serb and Themis Prodromakis: **A Tool for Emulating Neuromorphic Architectures with Memristive Models and Devices**. *2022 IEEE International Symposium on Circuits and Systems, in proceeding*, 2022
- Patrick Foster, **Jinqi Huang**, Alex Serb, Spyros Stathopoulos, Christos Papavassiliou, Themis Prodromakis: **An FPGA-based System for Generalised Electron Devices Testing**. *Scientific Reports*, 2022, doi: 10.1038/s41598-022-18100-3
- Patrick Foster, **Jinqi Huang**, Alex Serb, Themis Prodromakis, Christos Papavassiliou: **An FPGA Based System for Interfacing with Crossbar Arrays**. *2020 IEEE International Symposium on Circuits and Systems*, 2020, pp. 1-4, doi: 10.1109/ISCAS45731.2020.9180671.
- Patrick Foster, **Jinqi Huang**, Alex Serb, Themis Prodromakis, Christos Papavassiliou: **Live Demonstration: Electroforming of TiO2–x Memristor Devices using High Speed Pulses**. *2020 IEEE International Symposium on Circuits and Systems*, 2020, pp. 1-1, doi: 10.1109/ISCAS45731.2020.9180955.

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission

Signed:.. Date:..................

Acknowledgements

Firstly, I would like to thank my supervisors, Prof. Themis Prodromakis and Dr Alexander Serb, for leading me on the path toward becoming a qualified researcher. I would like to thank Prof. Michael Ng and Dr Nikitas Papasimakis as my administrative supervisors. I would also like to thank Prof. Geoff Merrett, Dr Adnan Mehonic, and Prof. Steve Gunn, my examiners in my final viva and during my PhD, for giving me helpful advice for conducting research and improving my technical writing. I would also like to thank my thesis committees for taking their precious time to organise the viva for me.

I would like to thank my lovely colleagues and friends, Dr Spyros Stathopoulos, Dr Firman Simanjuntak, Dr Shiwei Wang, Dr Sachin Maheshwari, Christos Giotis, Evangelos Moutoulas, Callum Aitchison, Patrick Foster, Jiaqi Wang, Yihan Pan, and Fan Yang for their professional and personal help and support.

On a personal note, I would like to thank my grandmother for letting me know she always has my back regardless of the situation. I would like to thank my friends Yi Yao, Qiang Gui, and Yitao Luo for giving me support and comfort even if with the distance.

Finally, I would like to thank myself for being courageous to move toward the direction I want without giving up, even with difficulties and challenges.

Jinqi

Acronyms and Abbreviations

AI: artificial intelligence
AMBA: Advanced Microcontroller Bus Architecture
ANN: artificial neural network
BP: backpropagation
DRTP: direct random target projection
ECM: electrochemical metallisation memories
FIFO: first-in-first-out
IMC: In-memory computing
IP: intellectual property
LIF: Leaky Integrated-and-fired model
LRS/HRS: low/high resistive state
LTP/LTD: Long-Term Potentiation/Depression
MAC: multiplication-accumulation
MSE: mean-square-error
NIC: Neuro-inspired computing
op-amp: operational amplifier
PSP: postsynaptic potential
RS: resistive state
SNN: spiking neural network
STDP: spike-timing-dependent-plasticity
STE: straight through estimator
STP/STD: Short-Term Potentiation/Depression
TCM: thermochemical memories
TPU: tensor processing unit
VCM: valence change memories
WTA: winner-take-all

Chapter 1

Introduction

1.1 Context

As a mainstream approach of artificial intelligence (AI), artificial neural networks (ANNs) have been widely employed in many disciplines, including pattern recognition [1, 2, 3], data mining [4, 5], system control [6], game playing [7], medical diagnosis [8, 9], financial data processing [10, 11], and machine translation [12], because of the generalisation ability and the computational power of deep neural networks. Loosely similar to biological neural networks, ANNs discard some biological properties to simplify the model. Spiking neural networks (SNNs), as new-generation ANNs [13], more closely mimic biological neural networks compared with their non-spiking predecessors [14]. Unlike traditional ANNs using continuous numerical values in computation, SNNs use spikes discrete in the time domain to encode information. In a time window, an SNN has one or more neurons firing and transmitting binary signals (all-or-nothing signals) called spikes through *synapses*. When a neuron receives spikes from neighbours, the membrane voltage increases or decreases depending on whether the synaptic potential is excitatory [15] or inhibitory [16]. The neuron then fires when the membrane voltage exceeds a certain threshold. After firing, the membrane voltage returns to a quiescent value, and the neuron cannot fire for a certain period.

There are some typical features of SNNs:

1. SNNs use binary spike trains to transmit information instead of continuous numerical values.

2. Temporal information, such as spike rates and the intervals between spikes, is applied in SNNs, to incorporate the binary spike coding method.

3. SNNs are highly event-driven because they only start calculation when spikes are received.

4. States of spiking neurons only depend on neurons' local information, such as firing states and spike timing.

5. Some SNNs involve recursion and recurrence, unlike the layer-to-layer propagation structure in ANNs. Therefore, the inference result is not given when propagated to the last layer as in ANNs, but when the first spike is generated.

These features bring SNNs great potential for lower energy consumption and better computational efficiency. Firstly, in traditional ANNs, the algorithm core is the dot product of input vectors and weight matrices. While in SNNs, a matrix multiplication turns into an addition because, in a single time step, each input can only be 1 or 0, resulting in less computational complexity [17]. Besides, conventional ANNs require computation for every input, while SNNs reduce the number of operations by only processing received spike events. The high sparsity of the spike trains not only further reduces energy consumption with fewer operations [17], but also increases robustness if the stochastic rate coding method is used [18]. To be more explicit, when using stochastic rate coding in SNNs, multiplication can be performed implicitly by using stochastic techniques. A typical case is where 2x stochastic bitstreams enter an AND gate. The statistics of computation dictate that the probability of a '1' at the output is the product of probabilities of '1' at the input streams, thus implementing multiplication in the probability domain. Naturally, the number of '1's within a time window along the output bitstream can be summed up to give an (approximate) reading of the probability number. In a similar manner, in SNNs, multiplication can be implemented implicitly, but more importantly, fault-tolerantly. Further, the scheme that neurons only process data from nearby splits the whole computation into smaller tasks, and each neuron only takes a small amount of workload. This operation scheme makes SNNs have higher fault tolerance because the work will be taken by nearby neurons if one is dead, though it requires re-training [19].

The typical way of implementing spiking neural networks in hardware is to process spikes and update neurons in discrete time steps. The simulation with a computing time step less than 1 ms can be regarded as meeting the real-time requirement because the biological time scale for cognition is 1 ms per inference. Some early works mapped spiking neural networks in parallel computers [20, 21, 22], clusters of multi-processor computer [23], hypercube supercomputer [24], or GPUs [25]. However, those designs hardly achieve the speed requirement of real-time simulation. Reference [25] is 1.5 times slower than real-time for a network with 100k neurons. Reference [21] can only run a real-time simulation in a network with less than 8k neurons. Reference [22] needs around 70 ms per time step to simulate an 8k-neuron network. Only reference [24] and reference [23] can achieve enough performance for real-time simulation of large scale spiking neural networks, with 0.1ms time step for running 64k and 100k neural network respectively [26]. For modern Von Neumann machines, given the limited computing resources, the network size and the speed need to be traded off. Therefore, the author predicts that for tasks that only require a small size of the network, the real-time requirement might be achieved. However, to scale up the network with a size similar to the brain, the runtime still overheads a lot.

When implementing SNNs, conventional computers have parallelism and memory bandwidth limits. From the system level, "neurons" are regarded as core computation units in SNNs, and neuron states need to be computed and updated concurrently within one time step [27]. The conventional sequential, centralised computation pattern can lead to long latency. From the architecture level, conventional Von Neumann architecture has separate computation units and memory. The bandwidth of communication between computation units and memory ('Von Neumann Bottleneck') [28] and the performance mismatch between them ('memory wall') [29, 30] are bottlenecks when frequent data movement is required. Compared with traditional ANNs, SNNs are based on a dynamic model, which requires storing both synaptic weights and internal potential. Since the SNNs model requires frequent data movement between memory and computation units, without sufficient memory bandwidth, large-scale SNNs implementation in conventional Von Neumann architectures is more communication-bounded than computation-bounded [26]. Von Neumann's bottleneck and memory wall also worsen the energy efficiency. For example, addition and multiplication of 32-bit floating-point operations take around 0.9pJ and 3.7pJ, respectively, in a 45nm technology with the power supply of 0.9 V, while accessing 64-bit data from DRAM takes 1.3-2.6nJ [30].

Some dedicated hardware designs have been delivered with a solution to place computing units and memory closely. Those designs include TrueNorth from IBM [31], Loihi from Intel [32], Neurogrid from Stanford University [33], BrainScaleS from University of Heidelberg [34], SpiNNaker from University of Manchester [35]. Those designs apply distributed memory, place neurons and synapses closer to diminish the communication cost, and support massive parallelism. With the new architecture, they show promising performance in SNN implementation. TrueNorth, Loihi, Neurogrid and SpiNNaker all achieve real-time SNN simulation with 16M, 128k, 1M, and 460M neurons, respectively. BrainScaleS even speeds up 10000 times faster than real-time with the support of 3.6M neurons. Dedicated neuromorphic hardware outperformed general-purpose computers in both speed and energy efficiency, despite the lack of flexibility in choosing neuron models and on-chip learning rules (if implemented) and the inconvenience of accumulating weighted spikes. Besides, FPGAs are also used to speed up the neural networks, for both spiking [17, 36, 37, 38, 39, 40] and non-spiking [41, 42, 43, 44, 45, 46] versions, because of the programmability, the sufficient resources and fine-grain parallelism.

In-memory computing (IMC) is a solution to problems caused by Von Neumann architecture. In this thesis, IMC is strictly defined as the computing method based on a 2D crossbar array. The computing method that places computing units and memory closely to reduce communication costs is called 'near-memory computing' instead. Instead of shifting data between memory and processing units, computations are performed where data are stored [47]. Based on a 2D array structure, efficient matrix multiplication is realised by storing one operand in a memory array while applying another operand to activate the rows of memory elements. The multiplication-accumulation (MAC) results are read from the array's columns. IMC is more efficient considering speed and energy than conventional Von Neumann hardware [48]. From the speed aspect, reference [49] shows that, in theory, the latency of accessing D bits data from a $\sqrt{D} \times \sqrt{D}$ SRAM array in a conventional way is D times as long as that of in IMC way in the worst case. From the energy aspects, reference [50] estimates the energy consumption of 4-bit MACs and 1024 dimensional vectors with 1-bit memory storage in 45nm technology. The total energy of memory accessing and computation is around 550fJ in the Non-IMC case, whilst the total energy is 50fJ in the IMC case. Design [51] also reports around 450 times better energy efficiency than Google TPU [52] ($\sim$866 TOPS/W versus $\sim$2 TOPS/W). References [53], [54], and [55] also successfully implement IMC in SNN hardware designs, showing IMC a promising approach to perform SNNs.

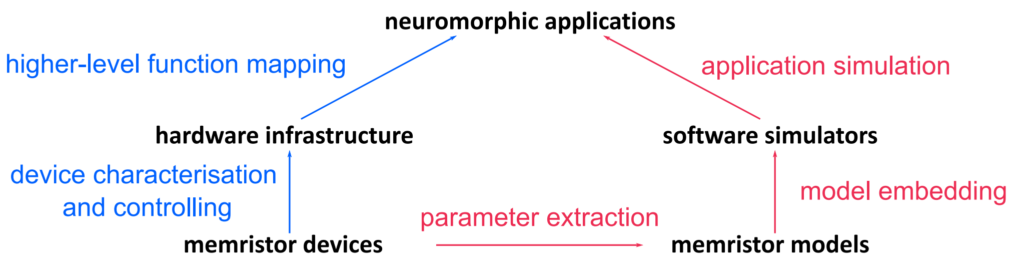


FIGURE 1.1: Pathways for applying memristive technologies to applications.

Memristors, firstly introduced by [56] as two-terminal circuit elements characterised by a relationship between the charge and the flux-linkage, are the fourth basic circuit element. Under this category, one important type is random-access memory devices (RRAM), cutting-edge memory that can be used for in-memory computing applications. Memristors are commonly organised in crossbar arrays, which can serve as matrix-vector multiplication accelerators [57, 58]. By setting one operand as input voltages from the rows and storing another operand as resistance in the array, the multiplication results can be easily calculated as currents along with the columns, directly following Ohm's law and Kirchhoff's current law. References [59], [60], [48], [61], [62], and [63] have performed matrix multiplication acceleration using memristor crossbar arrays in feedforward neural networks, convolutional neural networks, and long short-term memory. Compared with SRAM, memristors are better memory devices for implementing IMC-style SNNs, because of multi-bit storage ($\sim$6.5 bits [64]), high speed (85ps [65] vs. $<$10ns for SRAM [66]), low power consumption (10fJ [67] vs 50fJ in SRAM [49] and non-volatility [68]. Multi-bit storage allows a single cell to store more information, while non-volatility can store trained parameters without external memory if on-chip training is planned. In the example of [69], memristors have been used to accelerate SNNs. Besides, the inherent physical characteristics of memristors show high similarity to biological synapses to support the spike-timing-dependent-plasticity (STDP) learning rule [70], which is a popular learning rule in SNNs. Having the ability to implement STDP in memristor arrays makes it possible to train memristive spiking neural networks. References [71], [72], and [73] have applied on-chip learning in memristor-based SNNs.

1.2 Motivations and Challenges

Figure 1.1 illustrates two pathways for applying memristive technologies to neuromorphic applications. In general, neuromorphic designs can both mean designs that target revealing the mystery of brain functions, or that use bio-inspired architectures to perform AI tasks. In this thesis, neuromorphic applications are strictly defined as AI tasks, and neuromorphic designs are strictly defined as designs that use SNNs to perform AI tasks, unless stated. With memristor devices ready, we can first utilise hardware infrastructure to characterise and control devices, then map the higher-level functions in the hardware infrastructure to perform neuromorphic applications in hardware. Alternatively, we can also extract memristor parameters to model devices and embed the device models in software simulators to predict the outcomes of the applications before hardware efforts are dedicated. This pathway is for functionality validation and performance prediction mainly. The vast majority of works [71, 72, 73, 74, 75, 76] that focus on hardware implementations of the memristor-based spiking neural networks follow the first path, whilst there is also a work [77] following the second path to predicting the system functionality and performance in the pre-hardware design phase.

Despite multiple works successfully performing neuromorphic applications by taking the two paths, some facilities are still missing to complete the pathways to become mature guides for researchers to follow. Hardware infrastructure in the first path has two primary functions: (1) it acts as a testing and control system to handle devices, and (2) it delivers higher-level functions such as matrix multiplications with the incorporation of a memristor array to perform a neuromorphic application. For one thing, memristors, usually simplified as two-terminal tuneable resistors with non-linear switching dynamics, require specified characterisation, such as current-voltage sweep and incremental pulsing, for researchers to gain more intuition about the intrinsic physical properties. For another, memristors arranged in array structure further request a control and testing system that can provide read/write parallelism and high-speed data acquisition to match the parallelism of a neuromorphic design. Existing memristor characterisation systems either stay in the conceptual stage without physical experiments [78, 79] or do not provide read/write parallelism [80].

In the software simulator aspect, the software simulators in the second path undertake the task of modelling memristor devices, validating neuromorphic concepts, and elaborating device- and weight updating protocol-related details. Existing simulators [81, 82] focus more on the circuit level to emulate the hardware module behaviours in neuromorphic designs and to predict the performance specifications in chip designs. A simulator that (1) incorporates statistic device models, (2) sits at the algorithm level to validate the functionality of the neuromorphic concept before committing efforts in hardware design, and (3) provides step-by-step details in performing a neuromorphic task for investigation of system sensitivity is still missing.

In the application aspect, a vast range of neuromorphic applications with memristors in different disciplines have been achieved, including pattern recognition and image classification [74, 75, 76, 83]. However, memristor-based SNNs' usage is now limited to low-dimensional small-scale applications. This is because there are three major issues for expanding memristor-based SNNs in performing high-dimensional large-scale applications: high-dimensional inputs, costly training, and memristor non-ideality. To be specific, high-dimensional inputs require large memory allocation, which is costly in implementing memristor-based SNNs. For example, text classification, a particular area in natural language processing, often requires more than 10k dimensional inputs. Dimension reduction technology such as word embeddings, (e.g. GloVe [84] and word2vec [85]), are commonly used to map high-dimensional inputs to dense lower-dimensional representation for better space efficiency. However, no historical work has revealed the theoretical foundation of training a dimension reduction model such as a word embedding layer in a spiking neural network. Besides, both local (e.g. e-prop [86]) and non-local [87] gradient-based learning rules require accumulating the errors over a spike train window that is used to represent a single numerical value. This computation has poor speed and space efficiencies, especially when the spike train window is large and memristors are involved. Lastly, memristors introduce non-idealities due to the read noise and the write variation. These issues make training a memristor-based spiking neural network for text classification challenging.

In conclusion, to complete the pathways to apply memristor devices in neuromorphic designs, more practical efforts must be committed to constructing hardware infrastructure, software simulators, and methodology for specific applications.

1.3 Contributions

To address the above-mentioned needs, this thesis presents three designs to complete the two pathways shown in Figure 1.1. The first one is an FPGA-based digital interface that can be used to build a memristor characterisation and control system with 64-channel parallel read/write operations and high-speed data processing. The full control system not only acts as a testing tool for collecting memristor characteristics but also maps higher-level functions for device handling in neuromorphic designs. Furthermore, this system provides more powerful, versatile testing functionality for other two-terminal emerging memory devices beyond memristors. Resistor array handling and current-voltage sweep experiments are shown for functional validation of the digital interface.

The second design presented in this thesis is the first Python-based algorithm-level simulator, called NeuroPack, for memristor-based neuromorphic designs with an empirical memristor model [88]. With a wide range of selectable neuron, plasticity, and device models and the compatibility with user-defined models, NeuroPack assists users to quickly validating memristor-based neuromorphic concepts and monitoring the devices' behaviours during the training or the inference phase before serious efforts are committed in hardware designs. Besides, NeuroPack also stores internal variables, including membrane voltages, firing history, and weights for every single time step during the simulation. With a built-in result analysis tool, these data can be further used for analysis and visualisation so that users can investigate how intimately global parameters and critical design choices affect system performance. A 'predict-write-verify' loop also is proposed as the weight updating protocol implemented in NeuroPack to find the suitable pulsing parameter sets to trigger memristor state changes. This approach solves the challenge of precise memristor resistance update control due to the non-linear switching dynamics. Finally, an MNIST handwritten digit recognition task performed in a single-layer spiking neural network with the leaky integrated-and-fire neuron model [89] and winner-take-all structure [90] is showcased as a usage example of NeuroPack. With the final classification accuracy of 82.00% given the parameter settings, the accuracy degrades 1.55% compared with the accuracy achieved by the equivalent structure without memristive synapses.

The third design solved the issues of high-dimensional inputs, costly training, and memristor non-ideality in expanding memristor-based SNNs in high-dimensional large-scale applications by extending NeuroPack for a bespoke simulation framework with [91] to enable GPU compatibility and taking two paths to obtain trained spiking neural networks with memristor models: 1) by converting a pre-trained artificial neural network (ANN) to a memristor-based SNN, or 2) by directly training a memristor-based SNN. These two paths have two application scenarios: offline classification and online training. The first demonstration of a text classification task has been executed in memristor-based spiking neural networks with a data-driven memristor model [88] using the developed simulation framework. The task is to perform sentiment analysis in the IMDB movie reviews dataset [92]. By converting a pre-trained ANN to a memristor-based SNN and training the memristor-based SNN directly, the classification accuracy of 85.88% and 84.86% are achieved, respectively, with the equivalent ANN achieving a baseline training accuracy of 86.02%. From ANNs to SNNs and from non-memristive synapses to data-driven memristive synapses, comparable classification accuracy is achievable in simulation. In addition, investigations of the neural network sensitivity to global parameters such as spike train length, the read noise, and the weight updating stop conditions have also been given.

The contributions of this work are summarised below:

1. Developed and benchmarked an FPGA-based digital interface that allows channel-level parallelism and high-speed data processing for memristor array testing and control.

2. Designed the first algorithm-level simulator for memristor-powered neuro-inspired computing with selectable neuron, device and plasticity models, and elaborated the neural network sensitivity to device-related parameters with the demonstration of an MNIST handwritten digit recognition task performed in the presented simulator.

3. Demonstrated the first text classification task in the IMDB movie reviews dataset in spiking neural networks with a realistic memristor model by taking two different approaches to obtain trained neural networks, and explored system sensitivity to global parameters.

1.4 Thesis Outline

The thesis has six major chapters: chapter 1 presents the context of SNNs, their main features, their advantages over traditional ANNs, and the promising potential of memristor-based SNNs. Next, the current obstacles to developing memristor-based SNNs are introduced. Later, the contributions of the works aiming at addressing the previously mentioned issues are listed. These works will be the focus of this thesis.

Chapter 2 presents the theoretical background and historical designs of memristor-based neuromorphic applications and their relatives. Starting from the basic concepts of SNNs and memristors, historical efforts of neuromorphic hardware designs, including large-scale frameworks and low-power systems in CMOS, FPGAs, and memristors, are described. Next, research in software simulators for both spiking neural networks and memristor-based neural networks is summarised. Finally, the applications of text classification are shown as examples. As there is no existing memristor-based solution for text classification, the thesis introduces the mainstream solutions of using deep learning for text classification and typical memristor-based SNN applications to find out the existing challenges of utilising memristor-based SNNs in text classification tasks that this thesis needs to address. The needs for conducting the research presented in this thesis are concluded from all mentioned historical designs.

In chapter 3, an FPGA-based digital interface for memristor arrays is introduced. This work addresses the need to build a control system that can provide read/write parallelism and high-speed data processing for device characterisation and functionality delivery in a board-level neuromorphic design. The chapter starts with the introduction of the background, motivations, contributions, and objectives, followed by the design implementations of the system, from the system overview to specified modules. A resistor array handling and a memristor current-voltage sweep experiments are delivered to validate the interface's functionality. Finally, the control system built with this digital interface is compared with other similar works, and the conclusion is given.

Chapter 4 presents a Python-based algorithm-level simulator for memristor-based SNNs. This work aims at acting as a flexible tool for users to quickly validate the neuromorphic concepts before efforts are made in hardware designs and assisting users in exploring system sensitivity to specified parameters in simulation. This work provides a range of device, neuron, and plasticity models for users to choose from. The background, motivations, and contributions are firstly introduced to start the chapter. Next, the design implementations are displayed with a top-down workflow. Critical parts of the design, including the neuron models, plasticity models, memristor models, and weight updating scheme, are described. After that, the experiments of an MNIST handwritten digit recognition are presented to validate the simulator, and the investigation is delivered to find out how the device- and updating protocol-related parameters affect system performance.

In chapter 5, the demonstration of a text classification task in the IMDB movie reviews dataset is performed in a memristor-based SNN with an empirical memristor model. This example takes two different paths to obtain a trained memristor-based spiking neural network: one is converting a trained ANN to its equivalent memristor-based SNN, and another is training a memristor-based SNN directly. To do so, a simulation framework using memristor models is developed. The chapter is started with the background, motivations and contributions. Next, the methodology used to perform this text classification task is given, with critical design details elaborated. Finally, the experimental results with standard configurations and varying parameter values are displayed, and the result analysis is given.

Lastly, in chapter 6, the thesis is summarised, the achieved contributions of this thesis are concluded, and future perspectives toward AI hardware are exhibited.

Chapter 2

Memristor-based Spiking Neural Networks: A Review in Theoretical Background and Designs

2.1 Memristor-based SNNs Theoretical Background

This section will introduce SNN and memristor basics to give a theoretical background of memristor-based SNNs.

2.1.1 Spiking Neural Network Preliminaries

When implementing an SNN, there are some aspects to think about: (1) structure of the networks, (2) coding schemes, (3) neuron models, and (4) learning rules. In this subsection, those aspects of SNN will be introduced.

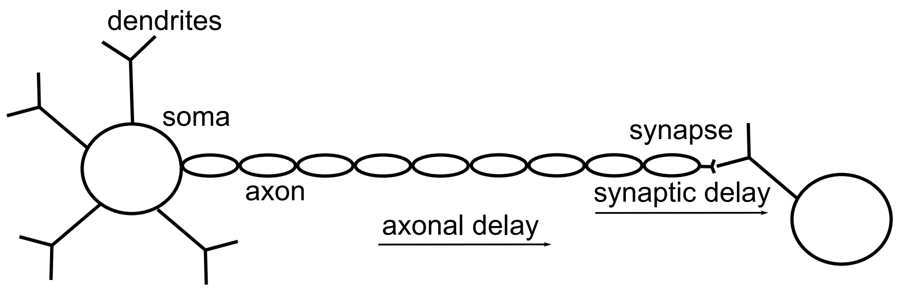


FIGURE 2.1: Biological neural network structure [17].

2.1.1.1 Network Structure

Figure 2.1 shows the structure of biological neural networks. Biological neural networks consist of dendrites, soma, axon, and synapses. Dendrites receive presynaptic spikes, and the soma accumulates the voltage. If a spike is emitted, it transmits through the axon and synapses to other neurons. SNNs are artificial neural networks that mimic biological neural networks. Therefore, a complete SNN also consists of those four parts, though the axon and dendrites sometimes are omitted in simulation to simplify the design. At a network level, there are also some commonly used types of SNN architectures, including winner-take-all networks [93] and liquid state machine [94].

2.1.1.2 Coding Schemes

Unlike ANNs using continuous numerical values in computation, SNNs deliver and process information through spike trains. Those discrete all-or-nothing binary signals with precise timing information allow SNNs to incorporate temporal information into the computation. That leads to the first question when implementing an SNN: which coding scheme to choose? There are two popular choices: rate coding and temporal coding. Rate coding is the coding method that only cares about the frequency of spike events, whilst temporal coding also conveys information through the timing of spikes. For example, spike sequences 0001100011 and 0101010101 deliver different information in temporal coding, whilst no difference is recognised in rate coding. Temporal coding can deliver more information, whereas rate coding is easier to implement. The vast majority of designs utilised rate coding because of its simplicity, whilst some designs insisted on temporal coding to provide more biological plausibility.

2.1.1.3 Neuron Models

An SNN is formed by a group of neurons that serve as computation units in the network. Therefore, choosing neuron models becomes a fundamental question in implementing an SNN. Generally speaking, a spiking neuron model works by changing membrane voltage, also called postsynaptic potential (PSP), according to the weighted sum of the incoming spike train. Notice that the potential changes can be bidirectional: the PSP can either inhibit or excite future firing, and the corresponding PSPs in these cases are IPSP and EPSP, respectively. Usually, IPSP can be regarded as the case of a negative weight in traditional

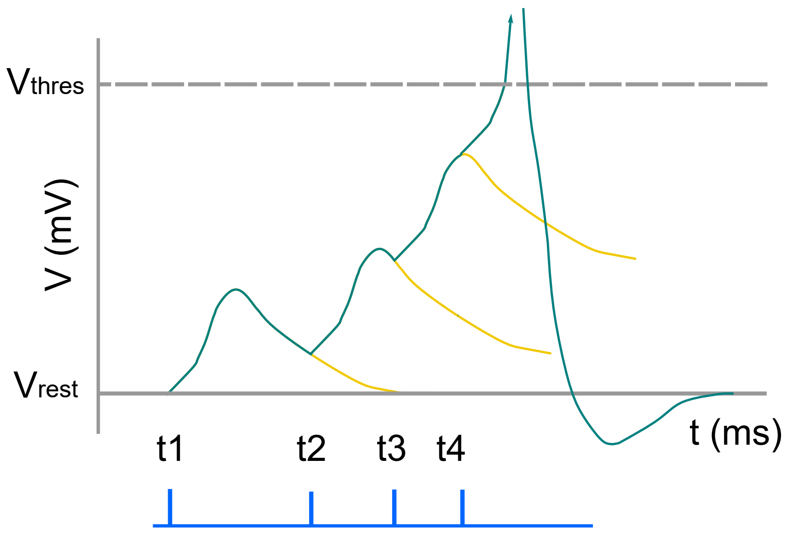


FIGURE 2.2: Membrane voltage changes when pre-synaptic spikes come. The image is from [95]. The x-axis is the time with the unit of ms, and y is the membrane voltage with the unit of mV. The blue spikes at the bottom show the incoming spike at time steps t1 to t4. When there is an incoming spike, the membrane starts accumulating, reaches the peak, and then decays. If the membrane voltage exceeds the threshold, a spike is generated, and the membrane voltage is reset.

ANNs. After the PSP exceeds the threshold, the neuron fires and the PSP is reset to zero. Figure 2.2 depicts the tendency of membrane voltage changes when spikes come.

Most spiking neuron models describe the relationship between input current response and output membrane voltage. The input current response expression is shown below:

$$u_i(t) = \sum_{j \neq i} w_{ij} x_{ij} + b_i \tag{2.1}$$

Where u_i is the synaptic response current, w_{ij} is the synaptic weight from neuron j to i, x_{ij} is the spike input, b_i is the constant bias. Among these types of neuron models, the most biological-like one is Hodgkin & Huxley model [96], which is based on a circuit model with resistors and capacitors shown in Figure 2.3, to mimic the effect of neurotransmitters between neurons. This model has a very complex form, shown below:

$$I_m = C\frac{dV_m}{dt} + G_l(V_m - V_l) + G_K(V_m - V_K) + G_{Na}(V_m - V_{Na}) \tag{2.2}$$

Where V_m, and I_m are the membrane voltage and current, respectively, C is the capacitance, G_K, G_{Na} are the potassium and sodium conductance, V_K, V_{Na}

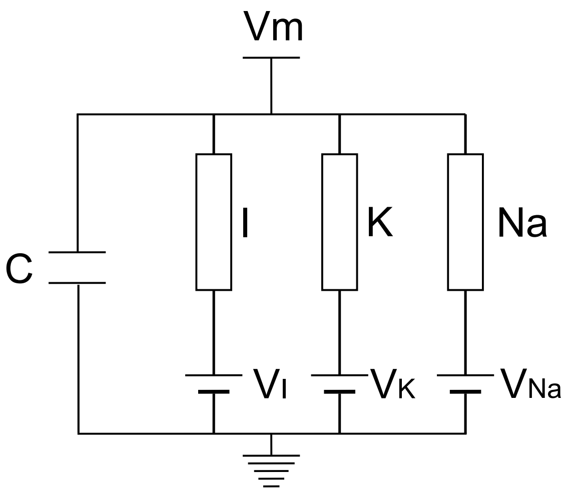


FIGURE 2.3: Equivalent circuit of Hodgkin & Huxley model [95]. V_m is the membrane potential. C represents the lipid bilayer. Voltage-gated potassium and sodium ion channels are represented by resistance R_K and R_{Na}, respectively. V_k and V_{Na} are potassium and sodium reversal potentials. R_I and V_I are leaky resistance and leaky reversal potential.

are reversal potentials [1] of potassium and sodium, and G_l and V_l are leak conductance and leak reversal potential. Here, G_K, G_{Na} are time and voltage-dependent with more internal variables.

Hodgkin & Huxley model successfully model the biophysical property of ion-channel-based neuron membrane voltage changes. However, due to the complex form, it is computationally expensive to apply Hodgkin & Huxley model in SNN simulation. A neuron in Hodgkin & Huxley model takes 1200 floating points operations per time step (typically 1 ms for real-time), and four internal variables are required to store in memory [97, 98].

The Leaky Integrated-and-fired model (LIF) is a simple neuron model that is computationally efficient. The mechanism of LIF can be represented as equations below [99]:

$$\dot{v}_i(t) = \begin{cases} -\frac{1}{\tau}(v_i(t) - v_{rest}) + u_i(t) & \text{if not firing} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Where u_i is the synaptic response current, v_i is the membrane potential, v_{rest} is the rest voltage usually assumed to be zero, and τ is the leaky time constant. The incoming spikes change the membrane voltage with the amounts according to the synaptic weights, and the neuron fires a spike if the voltage

[1] Reversal potential: the membrane potential when the ion channel is open

passes the threshold. After firing a spike, the membrane voltage resets to 0. Notice that the integration is leaky, given by the leaky term for membrane voltage. If there is no incoming spike, the membrane voltage will reduce gradually with the time constant τ. The integrated-and-fire neuron model is the simplified version of the Hodgkin & Huxley model, with the spikes' shape neglected. Spikes for the integrated-and-fire neuron model are always uniform so that only the time of appearance matters. The LIF neuron model only takes 4-5 FLOPs for one time step, and only one internal variable, the membrane voltage, is needed to store in memory [97, 98]. Some variants of IF or LIF can exhibit more biological attributes while keeping the computational efficiency. Those variants includes adaptive integrated-and-fire model [100], fractional-order leaky integrated-and-fire model [101, 102], exponential integrated-and-fire model [103], and adaptive exponential integrated-and-fire model [104], etc.

Izhikevich neuron model is another neuron model that is a good compromise between the biological plausibility of the Hodgkin & Huxley model and the computational efficiency of the integrated-and-fire neuron model [105]. It is also a simplified version of the Hodgkin & Huxley model while still keeping the shape of spikes to produce the spiking and bursting dynamics. The equation of the model gives below:

$$\begin{aligned} v^{'} &= 0.04v^2 + 5v + 140 - u + I \\ u^{'} &= a(bv - u) \end{aligned} \tag{2.4}$$

Where v is the membrane potential, u is the membrane recovery variable which represents K^+ and Na^+ ionic currents in the Hodgkin & Huxley neuron model. The term $0.04v^2 + 5v + 140$ was attained by fitting the observation of neuron dynamics. a and b are internal parameters for recovery variable u. It takes 14 FLOPs per time step, and only two variables are required to store in memory [97, 98]. Apart from these classic models, some are based on IF with stochasticity, such as the spike response model [106]. In general, if the design targets precise simulation of biological neural networks, Hodgkin & Huxley model should be applied; if the design focuses on computational efficiency, IF or LIF neurons should be applied; if the design trades off between computational efficiency and biological plausibility, Izhikevich neuron model, SRM, and IF variants can be utilised [97].

2.1.1.4 Learning Rules

Efficient adaptation and accurate inference of a learning system depend mainly on effective learning rules. For ANNs, the most common updating rule is gradient descent [87] with error backpropagation [107]. However, spiking neurons have internal state variables related to the sum-up of the input spikes, making the cost function non-differentiable. Therefore, the traditional supervised learning updating rules cannot be directly used in SNNs. SNNs updating rules are more inspired by natural neural processes and are more similar to Hebb's rule [2][108], which attempted to explain synaptic plasticity. Synaptic plasticity refers to the ability to adjust synaptic connections between neurons. This adjustment can be Long-Term Potentiation (LTP)/ Long-Term Depression (LTD) if effects last for hours or Short-Term Potentiation (STP)/ Short-Term Depression (STD) if effects only last for seconds or minutes. One of the most popular updating rules for SNNs is Spike-Timing Dependent Plasticity (STDP) [70], discovered in biological research. STDP tells us that synaptic plasticity is highly sensitive to the precise timing of the post-synaptic spikes related to pre-synaptic spikes. In other words, if the pre-synaptic spike comes earlier than the post-synaptic one, it will cause LTP; otherwise, LTD. The precise effect was given in [109] as below:

$$\Delta W_j = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t_i^n - t_j^f) \tag{2.5}$$

Here ΔW_j is the weight change of a synapse from pre-synaptic neuron j. t_i^n andt_j^f give the timing of post-synaptic and pre-synaptic spikes respectively, where n, f = 1, 2, 3 ...counts post/pre-synaptic spikes. $W(x)$ is an STDP learning window, of which a common choice is as below [109]:

$$W(x) = \begin{cases} A_+ exp(-x/\tau_+) & \text{for } x > 0 \\ -A_- exp(-x/\tau_-) & \text{otherwise} \end{cases} \tag{2.6}$$

Parameters A_+, A_- are scaling factors, and τ_+, τ_- are time constants which are usually $\pm 10ms$. Figure 2.4 shows this common choice of STDP window for induction of potentiation and depression characteristics with pre- and post-synaptic spiking timing.

STDP appears as a possible updating rule to underpin temporal Hebb's rule in SNNs. However, STDP is a timing-sensitive algorithm, and having timing as a

[2]Hebb's rule: A neuroscientific theory that synaptic efficacy increases when receiving persistent and repeated input stimuli.

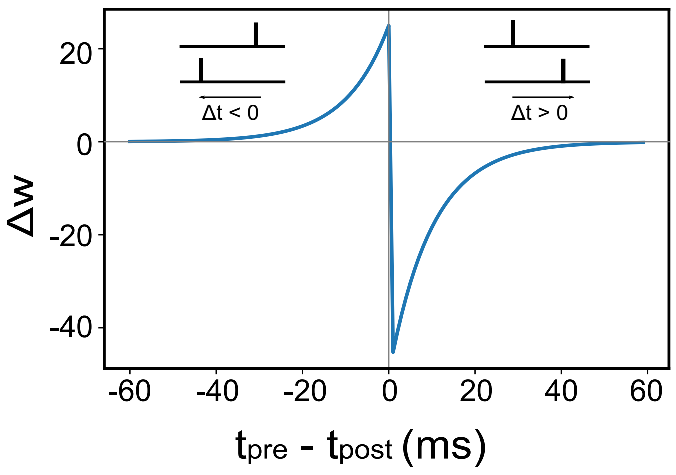


FIGURE 2.4: Asymmetric STDP learning window [70]. The plot is generated with $A_+ = 50$ and $A_- = 25$.

variable also needs precise control in hardware. Besides, STDP as an unsupervised learning rule requires adaptation to be explored to empower supervised learning tasks. Insufficiently development in learning algorithms underlies the main obstacle for making most of SNNs. It is believed that SNNs, compared with DNNs, are more of a long-term goal for researchers in neuromorphic area [95].

2.1.1.5 SNNs vs. ANNs

Here is a brief comparison in terms of energy consumption and speed between ANNs and SNNs: [110] showed the estimated energy consumption ratio between ANNs and SNNs is 1.72. When running ANNs and SNNs in the same digital platform, an inference in ANNs requires 2 FLOPs, and an inference in LIF-based SNNs requires 4-5 FLOPs. Given the limited computing resources, ANNs can be up to 2.5 faster than SNNs.

2.1.2 Memristor Preliminaries

2.1.2.1 Memristor Features

One critical point to emulate biologic neural networks is to use suitable devices to represent synapses. Non-volatility [68], multi-bits storage capability [64], low power consumption, and fast access time [111] make memristors good candidates [48]. First of all, the change of electric stimuli can switch the devices between low resistive (ON state) and high resistive (OFF state) [112], while

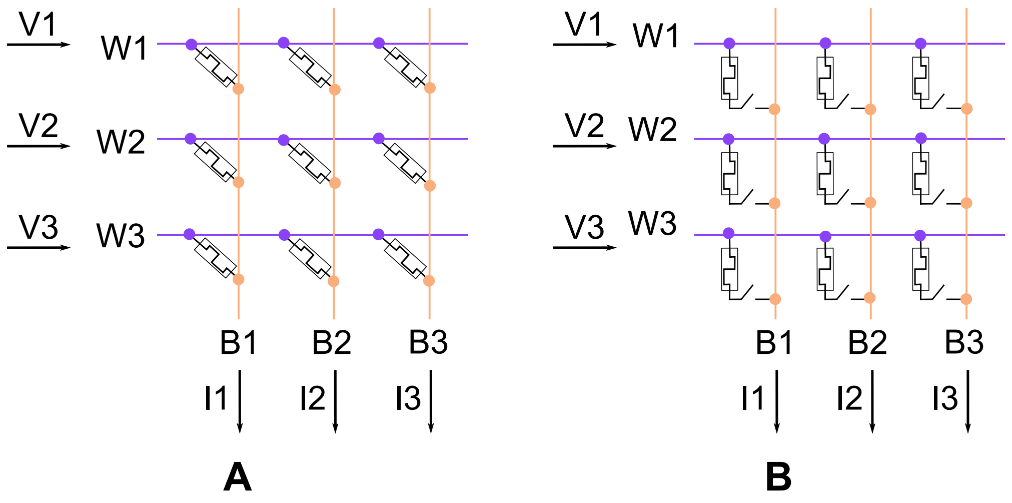


FIGURE 2.5: Memristor array without selectors **(A)** and with selectors **(B)**. Wordlines and bitlines are coloured in purple and orange, respectively. Notably, the crosspoints between wordlines and bitlines are not connected. When running in-memory computing in a memristor array with selectors, switches for selected devices are closed.

the resistive state will be retained after the voltage removes. Memristors' non-volatility can store weights with no extra power consumption, especially in a scenario of mapping and retaining pre-trained weights. Besides, a single memristor can store more than one bit with low power consumption. Reference [64] has reported 92 distinguishable states, which are equivalent to 6.5 bits when using digital memory, with energy consumption in the pJ - nJ range. Reference [67] mentioned ∼10fJ/operation for IMC in memristors, comparsed to 50fJ/operation estimated by reference [49]. Reference [113] further reported 115fJ for on-switching and 13pJ for off-switching for tantalum oxide memristors. Moreover, memristors also provide high-speed property, with switch time as fast as 85ps for nitride memristors [111], compared to <10ns read access time for SRAM [66]. Memristors are further specified into several categories: electrochemical metallisation memories (ECM), valence change memories (VCM), and thermochemical memories (TCM), based on working principles [114]: the bipolar ECM relies on the drift of the highly mobile ions conducting, the bipolar VCM switching is induced by voltage pulses, and the unipolar TCM relies on thermochemical mechanism, the change of the stoichiometry caused by a current-induced temperature increase. This thesis will only discuss VCM memristors.

2.1.2.2 Memristors and In-memory Computing

As two-terminal devices, memristors are suitable for matrix integration. With Directly applying Ohm's law and Kirchoff's current law, memristor crossbars are widely used to perform dot product [58], which is the core of an SNN algorithm. Figure 2.5 explains how a selectorless memristor crossbar forms a crucial part of a spiking neural network: rows (wordlines) and columns (bitlines) represent pre-synapse and post-synapse, respectively. Incoming voltage spikes, as one vector operand, are sent to wordlines, with synaptic weights stored as resistive states in memristors. The result of the weighted sum can be attained in bitlines as currents. Memristors are more commonly integrated into crossbars with selectors in series, as shown in the right figure of Figure 2.5 to get rid of the sneak path current, though increasing the difficulty in integrating crossbar in high density. With a crossbar structure, memristors can be used in in-memory computing with good energy efficiency. Table 2.1 displays the comparison of energy efficiency remarked by numbers of Tera-operations per Watt among Google's tensor processing unit (TPU) [115], a SRAM-based IMC design [116], and two memristor-based IMC designs [117, 118].

TABLE 2.1: Energy efficiency comparison of Von neumann machine, SRAM-based in-memory computing design, and memristor-based in-memory computing designs

Designs	Google's TPU[115]	[116]	[117]	[118]
Categories	Von Neuman	SRAM IMC	memristor IMC	memristor IMC
Energy efficiency (TOPS/W)	0.24-0.31	3.12	11	28

2.1.2.3 Memristors and STDP

As mentioned in Section 2.1.1.4, STDP is a popular learning rule for SNNs, especially for unsupervised learning tasks. However, due to the complex mathematics form, this learning rule is not hardware friendly. Reference [119] has given a CMOS-based STDP implementation with 30 transistors per plastic synapse. Previous research has shown that memristors can implement classic STDP learning rules as memristors have intrinsic characteristics that are highly similar to STDP [120]. Reference [71] proposed an effective approach to applying weight-dependent STDP in neuromorphic applications. This approach uses a low-voltage pulse below the threshold as a pre-synaptic spike (Figure 2.6 **(A)**). A bipolar pulse whose negative part exceeds the threshold is applied as a post-synaptic spike (Figure 2.6 **(B)**). In this way, a single pre-synaptic spike will not switch the state of memristors, while a single post-synaptic will cause long-term

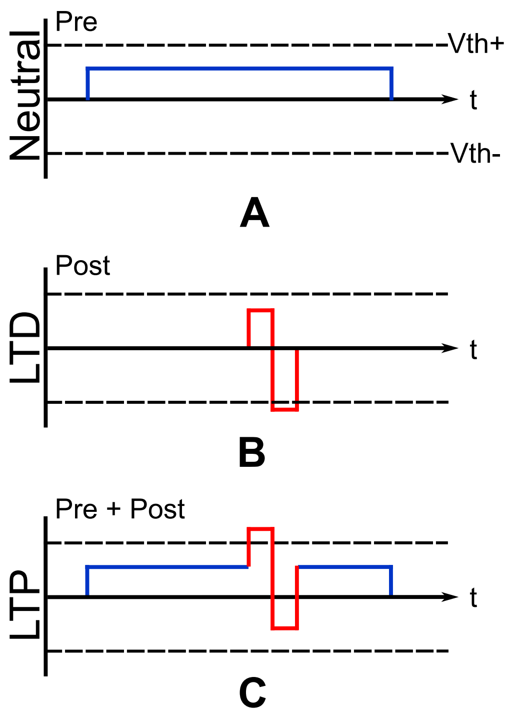


FIGURE 2.6: Memristors implement STDP [71]. This approach has been presented by [71]. Neutral **(A)**, LTD **(B)**, and LTP **(C)** are represented by a long, low-voltage under-threshold pre-synaptic pulse, a short bipolar post-synaptic pulse with the negative part exceeding the threshold, and the superposition of pre- and post-synaptic pulses, respectively.

depression (LTD). Suppose concurrent pre- and post-synaptic spikes are applied to the same memristor. In that case, the superposition will make the voltage exceed the positive threshold (Figure 2.6 **(C)**), which is equivalent to long-term potentiation (LTP). In other words, there are three possible spiking events to a single memristor: a pre-synaptic event, a post-synaptic event, and a combined pre-and post-synaptic event. They will lead to three outcomes respectively: neutral, LTD and LTP. The paper further looked into how the resistance state changed depending on the memristor's current state, which clearly shows an exponential tendency aligned with STDP. Based on this weight-dependent STDP characteristic, a general equation is summarized below:

$$\frac{\Delta g}{g} = POST \cdot [PRE \cdot f^{LTP}(g) - (1 - PRE) \cdot f^{LTD}(g)]$$

This approach enables memristor synapses to encode conditional probability so that inference and updating weight can be merged into one phase while training memristor-based SNNs, further empowering memristors to perform as plastic synapses in neuromorphic designs.

In summary, memristors bring great potential in implementing SNN designs by serving as synapses thanks to the low-power multi-bit storage capacity, crossbar structure, high-speed dynamics, non-volatility, and inherent physical characteristics similar to biological synapses. Despite great potential, larger sizes of memristor systems are still limited by device variability [67]. Besides, computing accuracy is also affected by device variations. So far, memristor-based in-computing ANN or SNN designs hardly achieve the same classification accuracy as their software equivalence [72, 74, 121, 122]. More efforts and exploration are still needed to make memristor-based neuromorphic become mainstream.

2.2 Neuromorphic Designs at a Glance

2.2.1 Hardware Frameworks

2.2.1.1 General Neuromorphic Design Strategies

The main goals of developing neuromorphic systems are performing AI tasks with the biological time scale and the power consumption, or understanding how brains work. These designs can either be potentially used in robotics [123], biomedical field [33, 124] and real-time online learning [125], or benefit from fast and low-power-cost information processing [31, 32]. In Section 2.1, a basic structure of biological neural networks has been given in Figure 2.1, with soma, dendrites, synapses, and axon. Inspired by biological neural networks, SNNs can choose to keep some of the biological attributes to balance design complexity and performance. In most SNN implementations, soma and synapses must be included, while dendrites and axons are optionally implemented. Hodgkin-Huxley neuron model [96] and Izhikevich's neuron model [105] use relatively complex mathematical expressions to model biological neural networks accurately, while they both require more hardware complexity. Leaky-integrate-and-fire neuron model [99] is more computational friendly, therefore LIF is a popular choice as the neuron model in SNN system designs [17, 38, 40, 53, 54, 71, 72, 126, 127]. Synapses are commonly represented as memory devices [53, 54, 71, 72, 73, 74, 75, 76].

In SNNs, information is encoded as spike rates together with the precise timing of arrival. The typical way to realise the temporal property of SNNs is to calculate and update neuron states in each time step, though this method brings the trade-off between precision and latency. Some designs [17, 31, 32, 38, 128, 129]

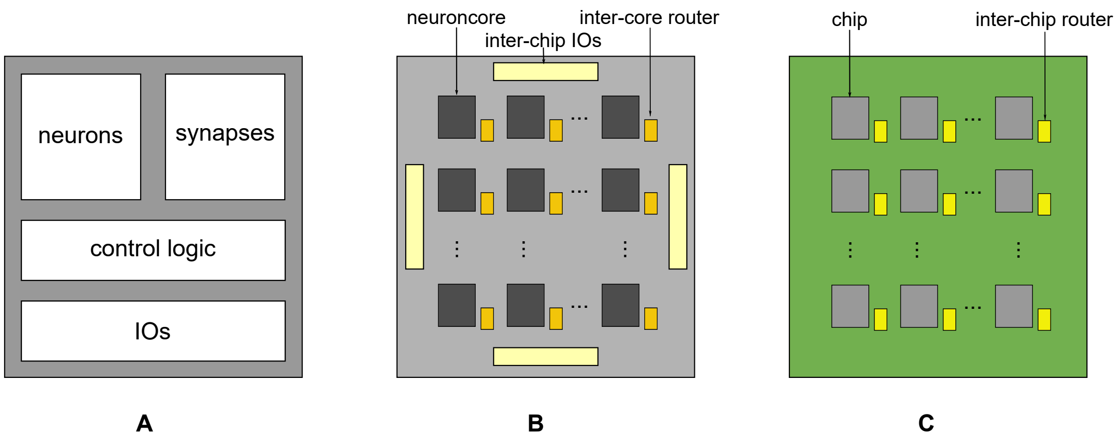


FIGURE 2.7: Neuromorphic system designs in different scales. **(A)** Neurocore. Neurons and synapses are placed closely to reduce communication costs. Each core has separate control logic so that multi-cores can work concurrently. **(B)** Chip. Multiple cores are integrated in this scale. Each neuroncore has an individual router, and a global NoC is used for cross-core communication. **(C)** Board. The image is inspired by [130].

also support fixed-axon and/or synapse delays to reduce information loss, despite the increased system complexity.

Neurons are regarded as the processing units in SNNs. In an SNN, a large number of neurons work separately and concurrently. Therefore, massive parallelism is the main challenge and the design goal when demonstrating an SNN regardless of the platform used. Memory utilisation and bandwidth are other issues that need carefully considered due to the large number of parameters used as synaptic weights. Some designs [53, 54, 71, 72, 73, 74, 75, 76] merge the computational units and the memory so that the data are processed in an in-situ manner to reduce the cost of accessing data to and from memory.

Figure 2.7 illustrates neuromorphic systems in different scale. This design hierarchy shows the same methodology for neuromorphic systems: distributing the memory storage and computation. Each neurocore contains part of neurons and synapses, so multiple neurocores can cooperate to speed up and save energy. Multiple neurocores in a chip and multiple chips on a board show parallelism at the chip and the board level, respectively.

Different targeted applications lead to different design choices. Large-scale neuromorphic systems aim at enabling and accelerating large-scale high-power datacentric computing tasks, whereas embedded processors are designed for mobile devices prioritising power consumption. Therefore, it is not fair to compare their performance directly. In the following subsections, typical designs for both large-scale neuromorphic systems and small-scale low-power designs are discussed separately.

2.2.1.2 Large-scale Neuromorphic Systems

Large-scale neuromorphic systems to be introduced in this subsection include application-specific integrated-circuits (ASICs) and application-specific instruction set processors (ASIPs) designs: TrueNorth [31] from IBM, Neurogrid [33] from Stanford University, Loihi [32] from Intel, SpiNNaker [35, 128] from University of Manchester, BrainScaleS [131] from Heidelberg University, multi-FPGA-based design BlueHive [129], and multi memristor crossbars based design INXS [132]. Table 2.2 gives the key characteristics of those seven designs.

TABLE 2.2: A comparison among large-scale neuromorphic systems. The table is inspired by [130, 133]

Platform	Neurogrid[33]	BrainScaleS[34, 131]	TrueNorth[31]	SpiNNaker[35]	Loihi [32]	BlueHive [129]	INXS[132]
Technology	analog	analog	digital	digital	digital	digital	mix-signal
Processor	Neurocore	HICANN[34]	4096 cores	18 ARM cores	128 cores	FPGA	512 memristor 32k crossbars
# Neurons	65k	512	1M	16k	128k	64k	128k
# Synapses	100M	100k	256B	16M	2M	64M	$\sim$ 226k
Power	150mW	1.3W	72mW	1W	N/A	N/A	2W
Power/neuron/Hz	2.3nW/Hz	0.25nW/Hz	72pW/Hz	62.5nW/Hz	N/A	N/A	N/A
Board	PCB	Wafer	PCB	PCB		Rack box	
# Processor	16	352	16	48		Up to 64	
Power	3W	500W	1W	80W		N/A [a]	
System		20 Wafers		600 PCBs			
Power		10kW		50kW			
Speed	1kHz	10MHz	1kHz	1kHz	N/A	1kHz	N/A
Chip Network	Tree multicast	Hierarchy	2D-mesh Unicast	2D-mesh Multicast	2D-mesh Unicast	High Speed Serial Links	2D-mesh Unicast
Neuron Model	Adaptive Quadratic IF	Adaptive Exponential IF	LIF	Programmable	Programmable	Izhikevich	LIF
Synapses Resolution	13b shared	4b	4b	Variable	1-64b	N/A	N/A
Plasticity	No	STDP	No	Pogrammable	Programmable	No	No
Feature Size	180nm	180nm	23nm	130nm	14nm FET	40nm	32nm
Applications	robotic control	bio simulation	cognitive appl.	bio simulation	machine learning	bio simulation	cognitive appl.

[a] No specific power information was given in the paper, but it was mentioned that BlueHive were more power hungry than SpiNNaker [129].

IBM's TrueNorth is a fully functional digital chip with 1 million programmable spiking neurons, 256 million configurable synapses in 4096 neurosynaptic cores, and intrachip networks for communication [31]. Each core has 104,448 bits of SRAM to store synapse states. Contrary to sequential, centralised conventional Von-Neumann architecture, TrueNorth uses a parallel, distributed design with a short-distance local connection and a long-distance global connection. Inspired by sparsity and event-driven property of biological neural networks, TrueNorth reduces power consumption by only transmitting spike events sparse in time between cores. Also, neurons' occasional defects do not dramatically disrupt system functionality because of fine-grained parallelism. The training process is not performed on-chip and needs the cooperation of specifically designed software. A video multi-object detection task with the input of 400 $\times$ 240 pixels 30-frame per second was executed to prove the functionality of performing AI tasks in TrueNorth.

Stanford University's Neurogrid is a neuromorphic system with subthreshold analogue neurons and synapses running in biological real-time (1kHz) [33]. The real-time operation allows Neurogrid to be applied in robotic control and biomedical applications such as controlling a prosthetic limb [133]. Each Neurogrid chip contains 16 neurocores, integrating more than 65k neurons and approximately 100M synapses. The neuron dynamics are implemented as an adaptive quadratic integrated-and-fire neuron model. The Neurogrid system does not use a mesh network like most other large-scale neuromorphic systems but applies a tree network, allowing deadlock-free multicast communication to interconnect neurocores. The system achieves 941pJ per synaptic connection.

Intel's Loihi is a fully integrated digital SNN chip supporting spiking neuron-based data encoding and processing with an off-chip communication interface and programmable learning rules [32]. Loihi has 128 neuromorphic cores, and each core contains 1024 spiking neurons. With the help of local connectivity and network-on-chip global connectivity, Loihi executes computational tasks with fine-grained parallelism. Loihi also has implemented several on-chip SNNs learning rules, making it more efficient to achieve on-chip learning. Loihi has been employed for the LASSO optimization task, showing this design is three-order of magnitude superior to a CPU-based solver in terms of energy-delay product.

University of Manchester's SpiNNaker is a massively parallel digital computer for large-scale spiking neural networks in biological real-time (1kHz) [35]. The SpiNNaker is a general-purpose spiking neural network system aiming at large-scale tasks and flexible implementation. This design contains 600 PCBs at the system level, and each PCB includes 48 processors. The interconnection among chips incorporates a 2D-mesh network with the ability to propagate the spike to multiple destinations. At the chip level, a SpiNNaker processing chip uses 18 ARM9 cores, each containing 32kbytes instruction memory and 64kbytes data memory. SpiNNaker does not have the best energy efficiency or the highest latency compared with other large-scale neuromorphic designs. However, it is the most reconfigurable because of selectable neuron models, synapse resolutions and learning rules for on-chip learning.

BrainScaleS from the University of Heidelberg is designed for accelerating tasks that take too long to execute in biological real-time [133]. BrainScaleS neuromorphic system delivers 512 adaptive exponential integrated-and-fire neurons and more than 100k synapses in one High-Count Neural Network (HiCANN), die [34], and 352 HiCANN dies are integrated into one wafer. Having 20 wafers in a small portable platform, BrainScale achieves massive parallelism operating 10k times biological speeds. FPGAs are used for high-speed serial communication between wafers [131]. BrainScaleS also supports the STDP learning rule.

BlueHive is a 64-FPGA spiking neural network simulator running in real-time with 64k Izhikevich neurons in each FPGA [129]. BlueHive uses 16 FIFOs to support 16 different synaptic delays with 1ms granularity. Parameters such as neuron states, weights, and delays are stored in off-chip memory. When accumulating neuron states, data are loaded from off-chip memory and sent back to memory after the calculation. FPGA-to-FPGA communication uses high-speed serial links together with PCIe connectors. The simulator of the 4-FPGA version achieved a speed of 162 faster than the CPU-version simulation.

INXS is a mix-signal chip aiming at improving energy efficiency and computational efficiency [132]. INXS uses modular and hierarchy tiled architecture with 512 256×128 memristor crossbars in each tile. One synapse consists of several memristors, each of which has 2-bit storage. After feeding the inputs to memristor crossbars and reading the output current through ADCs in bitlines, several bitlines are sent to a shift-and-add unit to combine the contributions of different bits of synapses to calculate the potential increment. The potential increment is then added to the old potential value, and the new potential is checked if exceeding the threshold. The neuron potential is retrieved/written back from/to SRAM. All steps but the crossbar processing are completed in the digital domain. INXS demonstrated 10.4× energy efficiency and 3129 × computational efficiency compared with TrueNorth.

2.2.1.3 Low-power Neuromorphic Systems

Small-scale neuromorphic systems are usually designed for mobile devices. Therefore, achieving high energy efficiency is the main design target. In this subsections, several small scale low power designs are discussed, including CMOS [53, 54, 126, 127], FPGA-based [17, 36, 37, 38, 39, 40], and mix-signal designs with memristor crossbars [71, 72, 73, 74, 75, 76].

For CMOS implementation, reference [126] designed a digital custom chip in 28nm technology, with spike schedulers, weight memory, and neurons. The

TABLE 2.3: Comparison of four CMOS small-scale low-power designs

Designs	[126]	[127]	[54]	[53]
Technology	Digital custom chip	Digital custom chip	Mix-signal custom chip	Mix-signal custom chip
Implementation	28nm CMOS	65nm CMOS	45nm CMOS Digital neurons + SRAM synapse crossbar	45nm CMOS Digital neurons + SRAM synapse crossbar
Training	backpropagation off-chip	weight dependent STDP on-chip	STDP on-chip	offline
Network	FC	FC	Hopfield network	Boltzmann machine
Coding scheme	Rate	Rate	Rate	Rate
Validation task	MNIST	MNIST	Pattern recognition	MNIST
Neuron model	LIF	LIF	LIF	LIF
# Neurons	1306	316	256	256
Synapse resolution	7b	24b	4b	1b
# Synapses	513M	128k	64k	256k
Latency	0.68us-10.9us	N/A	N/A	N/A
Power	∼70mW	Training: 104.12mW Inference: 91.30mW	N/A	N/A
Accuracy	98.7	∼90	N/A	89

input spike trains are fed into spike schedulers to decode the weight index. The weights are sent to the neurons to be accumulated after being fetched from the weight memory. The training is done with backpropagation in an off-chip fashion. This design achieved high accuracy (98.7%) for the MNIST task with around 70mW system power consumption. Reference [127] proposed a hardware architecture to perform SNNs with on-chip learning. Similarly, local buffers are employed to hold spike timing. The accuracy of performing classification on resized (16×16) MNIST dataset is around 90%, with the power consumption of 104.12mW for training and 91.30mW for inference. References [54] and [53] are both mix-signal custom chips in 45nm technology with digital neurons together with SRAM synapses crossbar. [54] and [53] integrated 256 neurons with 64k synapses and 256 neurons with 1024*256 synapses respectively. Each synapse consists of 8 transistors SRAM. The process is split into two phases: In the first phase, the rows of SRAM crossbar are selected when there are incoming spikes, and the weights are read through the columns to the neurons; in the second phase, the membrane voltages are checked if exceeding the threshold. [54] also achieves on-chip STDP learning. [54] validated the design with a Hopfield network to perform a pattern recognition task, and [53] employed a restricted Boltzmann machine to perform a digit classification on resized (22×22) MNIST with the final accuracy of 89%. Table 2.3 gives the comparison of four CMOS small-scale low-power neuromorphic designs mentioned above.

As for FPGA-based designs, references [36] and [37] proposed SNNs with the Izhikevich neuron model. They both supported a relatively large number of

TABLE 2.4: FPGA-based small-scale neuromorphic designs

Designs	[17]	[40]	[38]	[39]	[36]	[37]
Technology	FPGA	FPGA	FPGA	FPGA	FPGA	FPGA
Implementation	Input buffer + axon & synaptic delays	Synaptic filter	Event queue	Spike FIFOs	N/A	Event queue
Training	N/A	Backpropagation off-chip	N/A	N/A	N/A	N/A
Network	CNN	CNN	FC	FC	FC	N/A
Coding scheme	Rate	Temporal	Rate	Temporal	Rate	Rate
Validation task	RF automatic modulation	MNIST	MNIST	MNIST	N/A	N/A
Neuron model	LIF	LIF	LIF	IF	Izhikevich	Izhikevich
# Neurons	1006	N/A	65k	1394	1024	64k
Synapse resolution	N/A	N/A	16b	8b	8b	16b
# Synapses	N/A	N/A	16.78M	4.7M	1M	64M
Latency	N/A	0.52ms	236us	N/A	0.72-1.1us	N/A
Power	N/A	5.4W	1.5W	N/A	N/A	N/A
Accuracy	91.7	99.2	92	97	N/A	N/A

neurons and synapses. Reference [17] developed a streaming SNN architecture with fixed axonal and synaptic delays. Input buffers are used to store incoming event-driven spikes. The membrane voltages are initialised by the previous values stored in on-chip memory by multiplying the decay factor, and the weights are loaded if there are input spikes at that time step. After adding up all weights sequentially, the increment is added to the initialised membrane voltages, which is compared with the threshold afterwards. The design showed an application of radio frequency (RF) automatic modulation with an accuracy of 91.7%. Reference [38] developed an SNN system supporting 65k neurons per board with a system power consumption of 1.5W. The system is designed with three submodels: a LIF-based neuron model, a synapse model, and a fixed-delay axon model. Like other FPGA-based designs, the input events are stored in a queue. This design performed an MNIST classification task with an accuracy of 92%. Reference [39] built an alternative SNN based on temporal coding with 16 spike FIFOs. The system achieved an accuracy of 97% for the MNIST digit classification task. Reference [40] proposed a network of Infinite Impulse Responses (IIR) filters to implement SNNs with a temporal population neural coding scheme. The hardware implementation uses a layer-wise pipeline of processing elements that receive spikes from the previous layer, update synapse and neuron states, and send results in inter-layer buffers. Filter coefficients and synapse weights are stored in block memory, and the delay unit is implemented in shift registers. A digit classification task on MNIST performed in this system achieved an accuracy of 99.2% with a latency of 0.52ms and a system power consumption of 5.4W. Table 2.4 gives the comparison of those designs.

TABLE 2.5: Comparison of memristor-based small-scale neuromorphic designs

Designs	[71]	[74]	[72]	[75]	[76]	[73]
Technology	Mix-signal	Mix-signal	Mix-signal	Mix-signal	Mix-signal	Mix-signal
Implementation						
Neurons	Software/hardware	Software	N/A	Analog	Analog	Analog
Synapses	Memristor	Multi-memristive	Memristor	Memristor	Memristor	Memristor
Training	Weight dependent STDP	STDP	STDP	Hebb's rule	N/A learning	STDP
Network	WTA	FC	FC	FC	Hopfield	WTA
Coding scheme	Rate	Rate	Rate	Rate	Rate	Rate
Validation task	Pattern recog	MNIST	MNIST	Pattern recog	Pattern recog	Pattern recog
Neuron model	LIF	IF	LIF	IF	IF	IF
# Neurons	6	1044	834	38	12	6
Synapse resolution	N/A	∼4.3b	N/A	N/A	N/A	N/A
# Synapses	8	1.96M	39k	192	144	8
Latency	N/A	N/A	N/A	N/A	N/A	N/A
Power	N/A	N/A	N/A	0.577mW	N/A	N/A
Accuracy	N/A	>88.9	76.8	94.43	N/A	N/A

In memristor-based designs, memristors are used as synapses, and the multiplication of input spikes and weights is achieved by reading current from the post-synaptic side of the memristor crossbar as it is explained in references [71, 72, 73, 74, 75, 76]. Among those designs, references [71], [74] and [72] converted the synaptic current immediately to digital signals so that neurons can be implemented in software, while references [75], [76] and [73] designed analogue neurons. References [71], [74], [72], [75] and [73] also applied STDP or Hebb's rule by using the characteristics of memristors to perform on-chip learning. Those designs were validated through pattern recognition in either supervised or unsupervised fashion. Details can be found in table 2.5.

2.2.1.4 Summary of Hardware Frameworks

In summary, existing neuromorphic designs focus on dedicated hardware designs in different platforms, including ASICs, FPGAs, and memristor arrays. Among all, memristor-based neuromorphic systems have shown promising potential in low-power consumption usage scenarios (e.g. embedded systems). However, when utilising memristors in neuromorphic designs, another type of hardware infrastructure that conducts parallel device handling and higher-level function mapping from neuromorphic applications is equally important. There is no current work addressing the need for this type of hardware infrastructure, which motivates this thesis to fill the space for developing a digital interface for a memristor array control and characterisation system.

TABLE 2.6: Comparison of SNN and/or memristor-based NN Simulators (memristor-based NN simulators are colored in gray). The table is inspired by [134].

Simulators	Programming Language	training	SNN support	Usage
Brian [135]	Python		✓	general SNN simulation
Brian2GeNN [136]	C++, Python		✓	general SNN simulation
NEST [137]	SLI [a]		✓	general SNN simulation
NEURON [138]	a specified scripting language		✓	general SNN simulation
ANNarchy [139]	C++, Python		✓	general SNN simulation
Nengo [140]	Python		✓	neural behavior simulation
NeuCube [141]	unknown		✓	neural behavior simulation
BindsNet [142]	Python	✓	✓	AI purpose
SpykeTorch [143]	Python	✓	✓	AI purpose
snnTorch [144]	C++, Python	✓	✓	AI purpose
NVMSpice [145]	unknown	✓[b]		circuit-level simulation
NVSim [146]	C++, C	✓[b]		circuit-level simulation
NVMain [147, 148]	C++, Python, SystemVerilog	✓[b]		circuit-level simulation
MNSIM [82]	unknown	✓[b]		circuit-level simulation
NeuroSim [81]	C++, Python	✓		circuit-level simulation
TxSim [149]	Python	✓		circuit-level simulation
memTorch [150]	C++, Python			AI purpose
IBM toolkit [151]	C++, Python	✓		AI purpose

[a] SLI: a high-level scripting language *b*: Not naively supported

2.2.2 Software Frameworks

This subsection will introduce the software frameworks for emulating spiking neural networks and memristor-based neural networks. A comparison of SNN and memristor-based NN simulators is summarised in Table 2.6.

2.2.2.1 Spiking Neural Network Simulators

With the development of neuroscience and neuromorphic computing, a wide range of SNN simulation frameworks have been developed. These simulators are developed mainly for two purposes. One is to study brain functions and neural dynamics, and the other is to exploit SNNs in AI tasks.

The first type includes designs such as Brian [135, 152], Brian2GeNN [136], NEST [137], NEURON [138], ANNarchy [139], Nengo [140], NeuCube [141]. Brian [135] is a Python-based general-purpose SNN simulator providing a range of implemented neuron models. Users can also implement their neuron dynamics using the developed functions. The second version [152] adds the code generation functionality by separating the Brian core and a code generation engine that can generate code for different programming languages. Brian2GeNN further adds the functionality of running the Brian core in GPUs. Similarly, NEST [137] and NEURON [138], written in specified scripting languages, also aim to simulate individual neurons or large-scale SNNs. ANNarchy [139] is a similar framework to Brian2GeNN with a Python interface used to generate C++ code for rate-coded, spiking neural network simulation. Nengo [140] and NeuCube [141], on the contrary, focus on the simulation of neural behaviours and brain functionalities to enhance the understanding of how biological neural networks process information and deliver functions.

BindsNet [142], SpykeTorch [143], and snnTorch [144] belong to the second category. These simulators commonly extend the capability of mature ANN simulation frameworks such as Tensorflow [153] and Pytorch [91] to utilise tensor operations and allow GPU compatibility. BindsNet [142], SpykeTorch [143], and snnTorch [144] are all based on Pytorch, but they still have some differences. BindsNet is a flexible and general machine learning-oriented SNN library with a range of learning rules and neuron models, whilst SpykeTorch [143] focuses on restricted and optimised SNNs with time-to-first-spike coding method and the non-leaky integrate-and-fire neuron model only, and snnTorch utilises gradient-based learning rules to train SNNs.

2.2.2.2 Memristor-based Neural Network Simulators

Along with the research on utilising memristors to accelerate neural networks, the development of memristor-based neural network simulators has also grown. These simulators can also be divided into two categories: one provides circuit-level simulation with predicted specifications, and the other focuses on the conceptual simulation of architectures for AI applications with device non-ideality modelling.

Examples of the first category include NVMSpice [145], NVSim [146], NVMain [147, 148], MNSIM [82], NeuroSim [81], TxSim [149]. Among all, NVMSpice, NVSim, and NVMain are SPICE-based [154] that do not natively support inference or training. MNSIM and NeuroSim are designed for parametrised designs of memristor-based neuromorphic microchips. They collect information such as the crossbar array size, the ON and OFF resistances of memristive devices, the technical node, the read-out circuit module selection, etc., and then anticipate the layout area, dynamic power dissipation, latency, leakage power, and other performance indicators. NeuroSim supports both emerging non-volatile devices such as memristors and mainstream CMOS-based memory devices. Likewise, MNSIM employs a hierarchical, memristor-based neuromorphic computing system architecture. Finally, TxSim [149] represents a further refinement and includes DACs, memristor crossbars, and ADCs non-linearity when training on memristor crossbars.

In contrast, memTorch [150] and IBM Analog Hardware Acceleration Kit [151] (denoted as IBM toolkit for convenience) are examples of the second category. Written in Python, they extend Pytorch by including models of memristive devices. To be more explicit, memTorch enables the use of both linear and VTEAM memristors [155], as well as the estimation of non-ideal variations based on predefined distributions, and the IBM toolkit includes PCM models. In addition, both memTorch and the IBM toolkit are open-source and can run on CUDA. Notably, all memristor-based simulators mentioned above cover ANNs exclusively without supporting SNNs.

2.2.2.3 Summary of Software Frameworks

This subsection introduced software frameworks for SNNs and memristor-based neural networks. Notably, although many SNN simulators and memristor-based NN simulators exist, no memristor-based SNN simulators for algorithm-level simulation of machine learning tasks have been invented. However, this type of simulator is of increasing significance as more research attention has been drawn to incorporating memristor-based SNNs to improve hardware efficiency. This fact inspired the thought of inventing the first algorithm-level memristor-based SNN simulator presented in Chapter 4 to fill the vacancy.

2.2.3 Applications

In the application aspect, we compare deep learning applications and memristor-based SNN applications. We aim to find a reliable method for applying memristors for high-dimensional large-scale applications as deep learning solutions do. Therefore, this subsection reviews deep learning-based and memristor-based SNN applications and attempts to find out the challenges in using memristor-based SNNs to perform high-dimensional large-scale tasks.

2.2.3.1 Deep Learning Applications

Deep learning has been reported in applications spanning different disciplines, from image recognition to natural language processing, from recommendation systems to bioinformatics. This subsubsection gives a few typical examples of deep learning application summaries in Table 2.7. Reference [156] applied a CNN for image recognition tasks. Reference [157] adopted transformer [158], a self-attention based [159] deep neural network that overcomes the bottleneck of sequential data processing of RNNs by computing self-attention in parallel

TABLE 2.7: Deep learning and Memristor-based SNN Applications. Memristor-based SNN applications are coloured in grey.

Design	Synapses	Networks	Learning rule	Applications
[156]	digital	CNN	BP	image recognition
[157]	digital	transformer[158]	BP	natural language processing
[160]	digital	CNN	BP	recommendation system
[161]	digital	FC	BP	bioinformatics
[71]	memristors	WTA SNN	weight-dependent STDP	pattern recognition
[74]	multi-memristors	FC SNN	STDP	handwritten digit recognition
[72]	memristors	FC SNN	STDP	handwriteen digit recognition
[75]	memristor	SNN	Hebb's rule	pattern recognition
[76]	memristor	Hopfield SNN	N/A	pattern recognition
[73]	memristor	WTA SNN	STDP	pattern recognition
[162]	PCM	FC SNN	back-propagation	language modelling, music prediction
[77]	PCM model	SRNN	e-prop [86]	sequence prediction

for every token in the inputs, for natural language processing and achieved the state-of-art in eleven natural language processing tasks. Reference [160] used CNN for high-dimensional feature extraction in a recommendation system. Reference [161] used deep feed-forward NN with singular value decomposition (SVD) methods for dimension reduction in bioinformatics applications. In general, these applications usually require high-dimensional inputs and deep and large-scale neural networks, and sometimes need the assistance of dimension reduction models.

2.2.3.2 Memristor-based SNN Applications

We summarised typical memristor-based SNN applications in Table 2.7. Among all, references [71], [74], [72], [75], [76] and [73] employed memristor-based SNN in performing pattern recognition tasks. Two designs have demonstrated natural language processing tasks. Reference [162] used a PCM crossbar to accelerate SNN for language modelling and music prediction. Reference [77] simulated a spiking RNN for performing a sequence prediction task using e-prop [86], a newly proposed local gradient-based learning rule, with a PCM model.

2.2.3.3 Summary of Applications

This subsection introduced typical deep learning and memristor-based SNN applications. Deep learning has shown applications spanning across different disciplines with high-dimension inputs and large-scale networks, whilst memristor-based SNNs focus only on applications with low-dimensional inputs and small-scale networks. A possible reason is the lack of effective learning rules to train an SNN for extending the usage of memristor-based SNNs in high-dimensional large-scale tasks. Firstly, directly processing high-dimensional vectors is space- and computationally inefficient. Also, if a dimension reduction model, such as word embeddings (e.g. GloVe [84] and word2vec [85]), training in SNNs requires more theoretical support. So far, there is no such theoretical support provided by historical works. Besides, spike trains are generated to represent continuous values in SNNs. Gradient-based learning rules, commonly used in SNN training for supervised learning, require accumulating errors along the spike train to decide the update amounts of the continuous-valued parameters. This process is computationally inefficient. Finally, memristor non-idealities introduced from the read noise and the write variations make the training of the memristor-based SNNs even more difficult. These issues must be addressed to exploit the potential of the low-power computation of memristor-based SNNs for expanding the usage of memristor-based SNNs in more applications. Chapter 5 of this thesis aims to resolve these issues.

2.3 Summary

This chapter presents the theoretical background of memristor-based spiking neural networks from SNN preliminaries to memristor basics. Historical neuromorphic designs in hardware, software, and their applications have also been introduced. We noticed three critical types of designs missing to complete the pathways to apply memristors for neuromorphic applications: a control and characterisation system that handles devices and maps higher-level functions, an algorithm-level memristor-based SNN simulator, and a demonstration of text classification task utilising memristor-based SNNs. For the last design, there are also some theoretical challenges to overcome before the design can be developed. This thesis focuses on addressing the existing issues and developing missing designs to complete the whole picture for memristor-based neuromorphic applications. The following three chapters will introduce these three designs.

Chapter 3

An FPGA-based Digital Interface for Memristor Arrays

3.1 Introduction

Memristors have been used to exploit the potential in a wide range of applications, such as programmable logic [163], control systems [164], radio-frequency identification [165], signal processing [166], brain-computer interfacing [167], and neuromorphic computing [71, 72, 73, 74, 75, 76]. Among all, neuromorphic computing applications are the focus of this work. Benefiting from the intrinsic similarity to biological synapses [120] and the low-cost matrix multiplication computed by memristor arrays [57], many designs have employed memristors in neuromorphic computing. For example, reference [168] proved the possibility of using memristors in Bayesian inference; reference [71] explored the weight-dependent STDP and utilised this rule to train an unsupervised winner-take-all network, and reference [73] applied on-chip learning with STDP in memristors.

Meanwhile, the need for developing specified systems for testing and controlling memristor arrays is also increasing. For one thing, memristors are tuneable resistors with non-linear switching dynamics. This feature makes memristors require testing systems for specific characterisation, including current-voltage sweep and incremental pulsing, to gain more intuition of memristor behaviours. For another, memristors are commonly arranged in arrays, hence requiring light-weighted control systems with parallelism and high-speed data processing. So far, only a few instrumentation systems can provide these functionalities. Reference [78] reported a memristor-CMOS integrated array with high-voltage driver and voltage sensing systems, and reference [79] presented an on-chip characterisation system with 512×512 devices, but they both only have been simulated without being physically tested. Reference [80] is a 64-channel PCB-based characterisation tool for memristor arrays, but this work only allows read/write operations in a single channel at a time. A memristor array testing and control system that can provide both parallelism and high-speed data acquisition is still missing.

To address this need, this chapter introduces an FPGA-based digital interface that can be used to build a memristor array control system, which has two primary usage scenarios: 1) it acts as a characterisation platform for general testing, 2) and it serves as a control system allowing parallel reading and high-speed pulse programming in a board-level neuromorphic system. Beyond the usage for memristor devices, this control system can work as a much more powerful, general-purpose testing instrument for multiple devices. Experiments, including resistive array handling and memristor current-voltage sweep, have also been delivered to validate the functionality of this digital interface. In the end, the specifications of this system have been compared to similar systems, confirming the competitive performance this work can achieve.

With the aid of the digital interface presented in this chapter, research on incorporating memristors in neuromorphic designs can be accelerated by

1. easy testing of memristor characteristics,
2. channel-level parallelism in the read/write operations that improve matrix multiplications runtime from $O(n^2)$ to $O(n)$, specifically, 64 channel concurrent read and write need to be achieved,
3. flexibility of extending the system to implement a board-level neuromorphic architecture.

The major contribution of this work is summarised as developing and benchmarking an FPGA-based digital interface that allows channel-level parallelism and high-speed data processing for memristor array testing and control.

The chapter outline is as follows: in section 3.2, design objectives of the digital interface are described; section 3.3 gives design details from the high-level overview (subsection 5.2.1) to low-level implementations (subsection 3.3.2-3.3.4);

section 3.4 shows the experiments to validate the functionality of the digital interface; and section 3.5 summarises the chapter.

This work has led to these publications [169, 170, 171].

3.2 Design Objectives

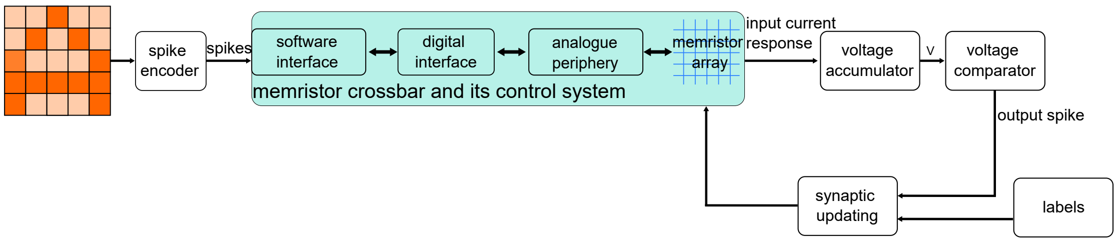


FIGURE 3.1: The topology of a hybrid SNN with memristor arrays. The icon in blue shows the structure of the memristor crossbar control system. The inputs are firstly converted to spikes through the spike encoder, and the input spikes are sent to the software interface of the control system. The digital interface bridges the communication between the software interface and the analogue periphery to handle memristor devices in the crossbar. After measuring the input current response from the crossbar, the voltage accumulator and the comparator calculate the membrane voltage and decide whether the neuron should fire an output spike. The output spike is then sent to the synaptic updating unit with labels to decide the weights used in the next iteration. Notably, the diagram is an abstract illustration of how the presented design can be used in a hybrid SNN. How the spike encoder, the voltage accumulator, the voltage comparator, and the synaptic updating unit are designed depends on the users' scheme, and it is out of the scope of this work.

A hybrid SNN topology is shown as an example (Figure 3.1) to illustrate what role the FPGA-based digital interface for memristor arrays plays in a PCB-level memristor-based spike neural network. A complete spiking neural network consists of several parts:

1. Encoding input spike trains;
2. Calculating the synaptic spike response current;
3. Updating internal variables such as membrane voltages;
4. Updating synaptic weights.

Firstly, the input data is converted to spike trains in software, and the software interface issues commands accordingly to the digital interface. The digital interface decodes the commands and sends input spikes as 0-or-1 binary pulses to the wordlines of the memristor array. Next, the memristor crossbar with its control system (digital interface and analogue periphery circuit) calculates the weighted sum of inputs according to the equation shown below:

$$u = \sum_{i}^{N} w_i x_i + b$$

Where u is the synaptic response current, w_i is the weight for the synapse i, x_i is the input from the synapse i, N is the number of synapses connected to the neuron, and b is the constant bias. In the hybrid system, calculating the synaptic spike response current is as simple as measuring the current flowing in the bitlines according to Ohm's law. After attaining the synaptic response current, the membrane voltages are updated according to the chosen neuron model. If the leaky integrate-and-fire neuron model is used, the new membrane voltages are calculated by adding the synaptic response current to the decayed current membrane voltages. After that, the magnitudes of the membrane voltages compared to the threshold determine if spikes should be generated, and post-spike membrane voltages are reset to 0 according to the comparison results. This step usually happens in the software, whereas an FPGA can also be the alternative if targeting more parallelism. Also, if output spikes are generated or not is regarded as inference results. Finally, the inference results are compared to the ground truth for supervised learning, the new weights are calculated in either software or an FPGA, and the digital interface sends pulses to trigger memristor resistive state updates. Therefore, the digital interface is designed to achieve the following functionality:

- Receiving, storing and decoding the commands sent from the software interface.
- Controlling the analogue periphery circuit to read and write the memristor array in a parallel fashion.
- Storing measurement results and fetching them according to commands issued by the software interface.

The following sections will explain how the functionality is achieved.

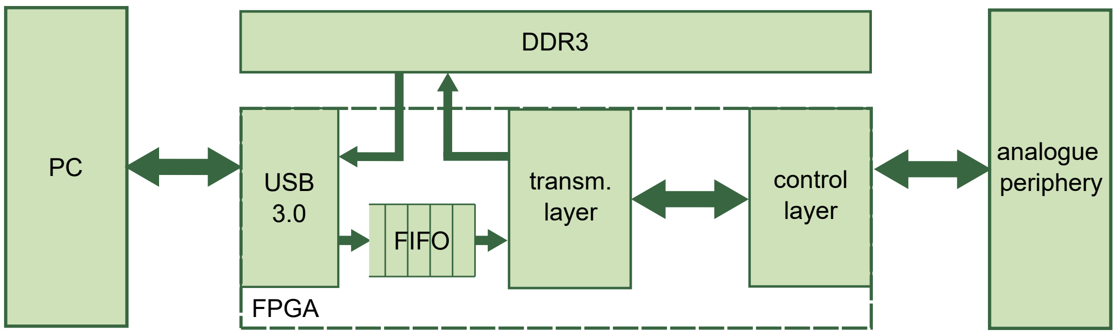


FIGURE 3.2: Digital interface of memristor crossbar control system shown in the dashed box. Arrows show the data flowing directions. The USB 3.0 IP is provided by Cesys [172]. The FIFO IP [173] and the DDR3 control IP [174] are provided by Xilinx.

3.3 Design Implementation

3.3.1 Design Overview

The digital interface is designed for a memristor array control system. The concept diagram of the memristor crossbar and its control system is shown as the blue icon in Figure 3.1. The control system is composed of a software interface, a digital hardware interface implemented in an FPGA, an analogue periphery circuit, and a memristor crossbar array. This chapter only focuses on the digital interface, which is based on Cesys EFM-03 development board [175]. The analogue interface and the software interface shown in Figure 3.1 were designed by other researchers.

The concept diagram of the hardware digital interface can be found in Figure 3.2. The software interface sends user commands to the digital interface through USB 3.0 to send voltage pulses to or read out voltages from selected memristor devices (in basic write/read operations). Those two basic operations are equivalent to sending pre-synaptic spikes and reading out synaptic response current in conventional spiking neural network hardware. In a basic write operation, voltage pulse amplitudes, pulse widths and target device IDs are encoded into commands and sent from the software interface through a USB3.0 intellectual property (IP) core. A first-in-first-out (FIFO) buffer is connected between the USB3.0 IP and the transmission layer to match the speed of data transmission and data processing. The transmission layer in the digital interface decodes commands into control signals used in the control layer to directly drive the analogue periphery circuit. In a basic read operation, the software interface first sends read commands with parameters such as read-out

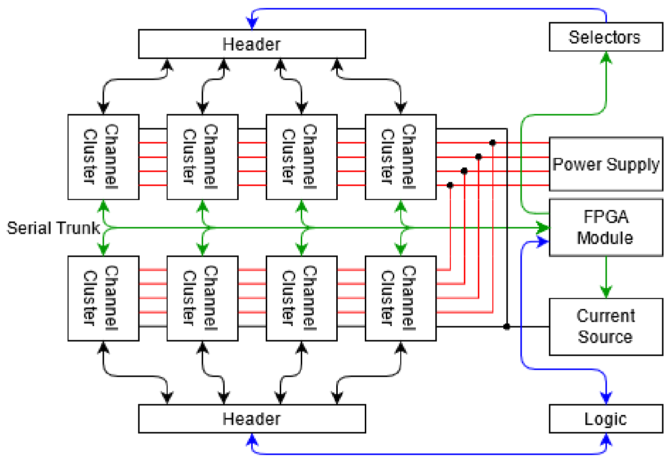


FIGURE 3.3: Schematic of the analogue interface. The figure is from the author's publication [169].

voltages and target device IDs. The transmission layer decodes the commands and enables the control signals for the control layer to start the read-out. Read-out results are stored in DDR3 off-chip memory. The implementation of the FPGA interface will be explained in the next section. The digital interface is realised in a Xilinx 7-serial FPGA. The USB 3.0 IP and the DDR3 controller IP in the digital interface are provided by Cesys [172] and Xilinx [173, 174], respectively.

3.3.2 Control Layer

The control layer drives the analogue circuit using decoded information from the software interface. Therefore, the functionality of the control layer is decided by the analogue periphery circuit. Before looking into the control layer's design details, understanding the analogue interface's concept structure helps determine what functionalities are expected in the control layer. The high-level design of the analogue interface and the analogue circuits in each channel are displayed in Figures 3.3 and 3.4 for clarity. Figures are exported from the paper [169] where the author worked as a joint author. This controller is designed for a 32×32 memristor crossbar array with 64 channels, each of which includes an operational amplifier (op-amp) to form a trans-impedance amplifier with different selectable feedback resistors. Channels are grouped into 8 clusters, each of which consists of 8 channels by using one 8-channel ADC chip, one 16-channel DAC chip, and one 88-bit switch chain. Besides, there are also global selectors, arbitrary DACs, current source controllers, and digital IOs to achieve flexible control.

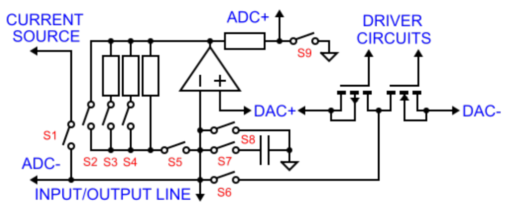


FIGURE 3.4: Schematic of the channel circuit in the analogue interface. The figure is from the author's publication [169].

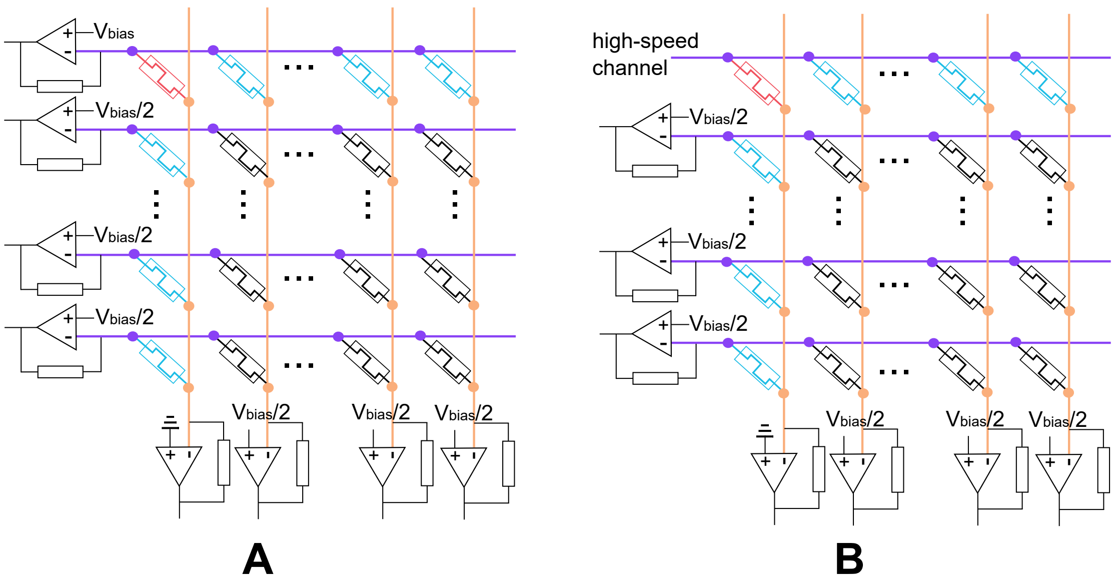


FIGURE 3.5: Read **(A)** and write **(B)** operation in a selectorless memristor crossbar. Wordlines (in purple) and bitlines (in orange) are not intersected. The red device is the selected device, the blue devices are half-selected, and the black devices are non-selected.

In a basic read operation for a selectorless array, a low bias voltage that cannot change memristor states is applied by DACs to the selected row. The selected column is grounded so that the current fed through the trans-impedance amplifiers in the column can be calculated through the feedback resistor value and the output voltage (Figure 3.5 **(A)**). There are three selectable feedback resistors (820 Ω, 110 KΩ, and 15 MΩ) in a single trans-impedance amplifier to accurately read back a wide range of voltages. Unselected devices will be connected to half of the bias voltage from both terminals so that there is no voltage difference between the two terminals. However, in a selectorless array, there are half-selected devices (which can be seen in Figure 3.5 **(A)**). Those in the same column with the selected device will contribute currents in the read-out. Therefore, the currents in rows with those devices will also be measured and stored. We subtract the measured current in half-selected devices from the measured current in the selected devices to get the accurate current flowing in the selected device. In a

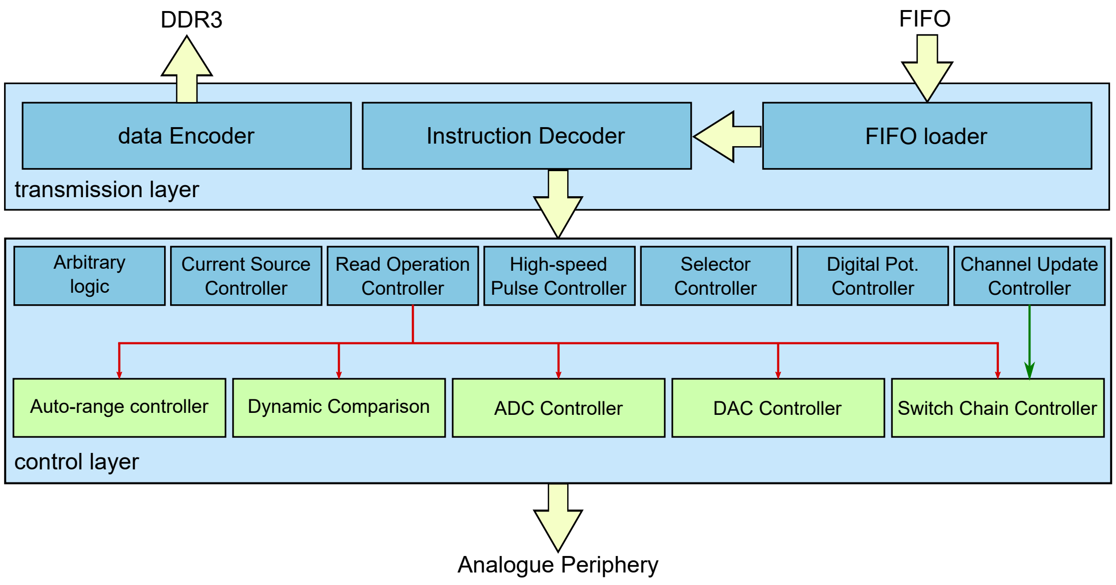


FIGURE 3.6: Hierarchy of the transmission layer and the control layer. The auto-range controller, the dynamic comparison, the ADC controller, the DAC controller, and the switch chain controller are driven by the read operation controller (shown as the red arrows). The switch chain controller can also be driven by the channel update controller (shown as the green arrow).

selector-based array, the half-selected issue is solved by activating the selector line of the selected device.

In a basic write operation, selected bitlines are still grounded, and selected wordlines connect high-speed channels, which use switches to achieve short pulses (Figure 3.5 **(B)**). Two DACs per channel are needed to apply bipolar pulses. Also, arbitrary logic configures flexible voltage references, selectors, and programmable current sources. Therefore, all components (DAC clusters, ADC clusters, high-speed channels, control switch chains, arbitrary logic, selectors, and current sources) in the analogue periphery circuit need the control of the digital interface.

Based on the structure of the analogue periphery circuit, the design diagram of the transmission layer and the control layer is decided as shown in Figure 3.6. One of the objectives of the digital interface is to maximise parallelism with reduced communication costs. To achieve this objective, each module in the control layer only controls one cluster. In the transmission layer, the data bus is connected to the PC to transfer final data results calculated in FPGA to benefit from the parallel computation power of FPGAs. The design uses synchronous communication to reduce communication costs and avoid contention. The design follows a bottom-up design flow: at the bottom are the lowest-level modules, including auto-range modules, dynamic comparison modules, ADC controller, DAC controller, high-speed pulse timer, and switch chain controller, for directly driving analogue periphery circuits. Among those modules, each auto-range controller and dynamic comparison only control one channel, while each ADC controller, DAC controller and switch chain controller control one cluster. The lowest-level modules are directly driven by higher-level modules, such as the read operation controller and the channel update controller, according to commands decoded from the transmission layer.

3.3.2.1 DAC Controller

DACs used in the analogue interface are DAC81416, produced by Texas Instruments. DAC81416 chips are 16-channel, 16-bit high voltage DACs [176]. DAC81416 chips use standard SPI protocol, with SCLK, SDI, SDO, and CS serving as the serial clock, serial data output, serial data input and serial data enable, respectively. Configuration registers and DAC data registers in DAC81416 chips are responsible for setting global configuration and storing voltage values. DAC register reconfigurations are needed before data are sent to DAC data registers, as default values do not suit the working conditions of the analogue interface. Please refer to Appendix A, section 1, for details.

According to the datasheet, the normal SPI transmission that DAC81416 chips support is a 24-bit access cycle. It contains a read or write command bit, a don't-care bit, a 6-bit register address, and 16-bit data. DAC data registers have addresses from 10h to 1Fh. There are two-stage operations to use the DAC chips: the configuration stage and the DAC performance stage. After power-up, DACs need to be configured properly before going into the DAC performance stage. Transmission in the configuration stage is a normal stand-alone 24-bit SPI communication. However, as all DACs channels will generate different voltages (V_{bias}, $V_{bias}/2$, V_{pulse} or 0) in read or write operations, updating all 16 channels is not efficient in a normal transmission mode. Another transmission mode, called streaming mode, is more efficient in feeding a large amount of data. By holding the CS signal low and continuously shifting data into DACs, streaming mode only needs a base address to shift more than one 16-bit data package into DAC data registers.

The DAC clusters control module is designed to have a configuration stage

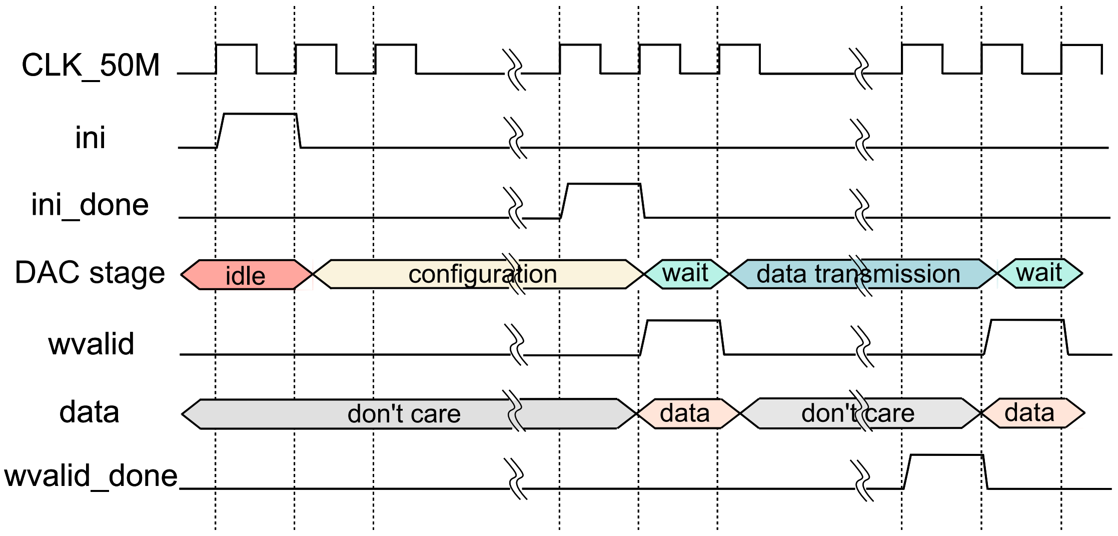


FIGURE 3.7: DAC control module protocol. Different colours show different stages.

working in normal transmission mode and a data performance stage in streaming mode. Figure 3.7 shows the protocol used in the control module. The control module is working under a 50MHz clock. There are four states - *idle*, *configuration*, *wait for data*, *data transmission* - in the control modules. Initially the program stays in *idle* state until the higher-level control module sends an *ini* signal to start the configuration. After shifting configuration data into configuration registers in DACs, the module generates an *ini_done* signal sent back to the higher-level control module. Only when an *ini_done* is received can the higher-level module starts sending data to DAC data registers. Data sent to data registers must be valid when a *wvalid* is valid. Similarly, when data transmission finishes, the module sends back a *wvalid_done*; after that, another *wvalid* is sent to start another data transmission operation. Table 3.1 summarises signals of a DAC81416 control module and how they work.

There are two DACs used in each crossbar channel, so to configure a 32×32 memristor crossbar, 128 DACs channels in 8 chips are needed. There are eight control modules described in previous paragraphs. They can be tied together to work simultaneously or independently to achieve flexible control.

3.3.2.2 ADC Controller

ADCs used in the analogue interface are LTC2358 produced by Analog Devices. LTC2358 is a 18-bit 8-channel differential ADC with working range up to ±10.24 V [177]. The LTC2358 supports SPI CMOS serial interface through SCKI, SDI, SDO (0, ..., 7), CS serving as the clock, serial data input, serial data output, and

TABLE 3.1: DAC81416 control modules signals

signals	signal type	description
ini	input	signal from higher-level control module to trigger the configuration
ini_done	output	signal to higher-level control module to indicate the finish of configuration
wvalid	input	signal from higher-level control module to trigger a data transmission operation
wvalid_done	output	signal to higher-level control module to indicate the finish of data transmission
data	input, 128-bit bus	16-channel 16-bit data that will be sent to DAC data registers

serial data enable. LTC2358 has a 24-bit softspan code, a code that can be sent by users, to select the input range for all eight channels, and the softspan needs to shift into the chip before every conversion. LTC2358 operates in two phases: the acquisition phase and the conversion phase. During the acquisition phase, chips simultaneously start sampling in all channels (if not disabled) by pulling up CONVST (0, ..., 7) signals. After that, BUSY (0, ..., 7) signals go high and back low when the sampling finishes. Holding CS signal low starts the conversion phase when 24-bit per channel conversion results, including 18-bit conversion results, 3-bit channel id, 3-bit softspan, shift out from SDO (0, ..., 7) signals. Meanwhile, 24-bit softspan code (3 bits per channel) for defining input ranges of next conversion shifts into SDI at every rising edge of SCKI. A complete conversion contains acquisition and conversion phases, so ADCs always need to switch between these two phases.

The analogue interface uses four of the eight data outputs lanes, meaning SDI takes 24-bit softspan code and SDO outputs 48-bit data results for two channels. The softspan code appears twice on SDI to match data lengths in SDI and SDO. Another issue is that the first conversion uses the last softspan code, which may cause some errors as the softspan may be outdated. To address this issue, another operation mode which only has the conversion phase is implemented in the module. This operation mode does nothing but starts a 24-bit data transmission to update the softspan we want. After updating the softspan, the state machine can be in the normal two-phase operation mode until the next softspan needs to be updated.

A finite state machine is used to realise the working scheme proposed above.

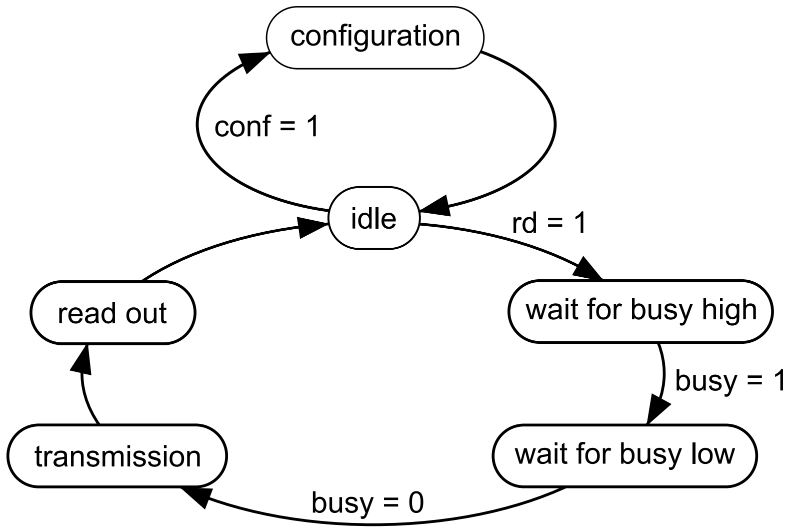


FIGURE 3.8: ADC control module state machine

Figure 3.8 shows how exactly the finite state machine of LTC2358 control module is designed. First, let us look into the structure of the state machine. After powering up, the state machine starts from the 'idle' state. When a valid *conf* control signal arrives, the state machine goes to the 'configuration' state to update softspan. After the softspan configuration, a *conf_done* signal is sent back to the higher-level control module, and the state machine returns to 'idle'. When a valid *rd* control signal arrives, the state machine goes to a full two-phase operation mode which will do both 'conversion' and 'acquisition'. In the design, this operation mode can be represented by 'wait_for_busy_high', 'wait_for_busy_low', 'transmission', and 'read out' four states: In the state 'wait _for_busy_high', an active CONVST signal is sent out to trigger the sampling. After a high BUSY signal is detected, the state machine goes to the 'wait_for_busy _low' state, which waits for the BUSY signal to go low. The low BUSY makes the state machine jump to the 'transmission' state. There is a time-out counter in the 'wait_for_busy_low' state to avoid the state machine being locked up. If no valid low BUSY signal is detected when the time-out counter hits the maximum possible value, the state machine still starts the 'transmission' state. The 'transmission' state starts shifting conversion results from outputs, and the 'readout' state updates the data register. After that, a *rd_done* signal is sent back to the higher-level control module, and the state machine goes back to 'idle', waiting for the next valid control signal.

Table 3.2 gives control signals in LTC2358 control module, and the whole protocol can be referred to Figure 3.9. There are four main control signals: *conf*,

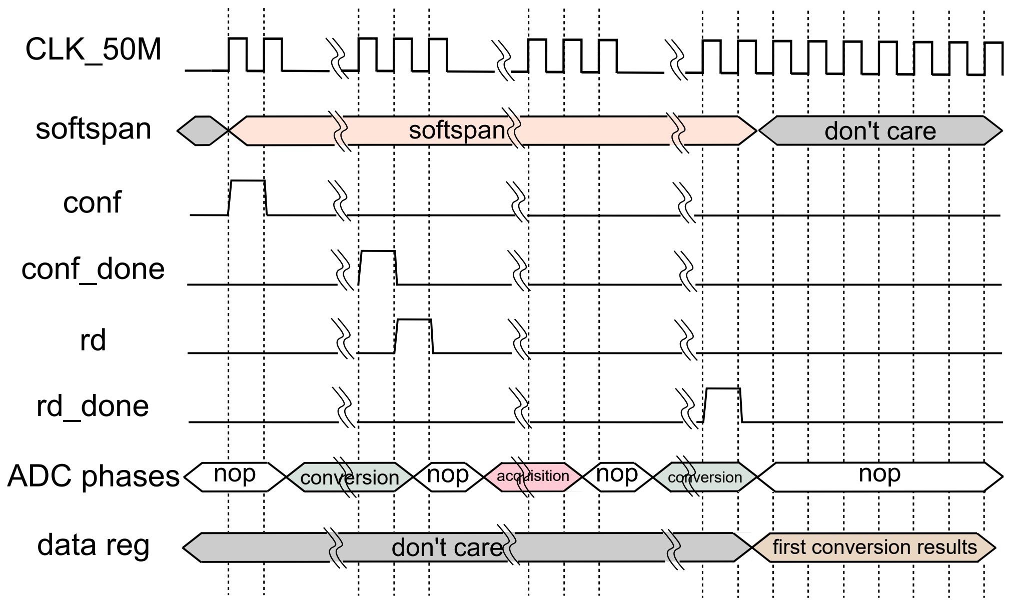


FIGURE 3.9: ADC control module protocol. different colour show different stages.

conf_done, *rd*, and *rd_done*. *conf* and *rd* are input signals generated from higher-level control modules, and *conf_done* and *rd_done* are outputs signals to higher-level control modules. Signal *conf* triggers a update on softspan and signal *conf_done* indicates the end of this operation. Signal *rd* triggers a normal two-phase operation including conversion and acquisition and signal *rd_done* indicates the whole operation is done. These four signals are all high-active pulses. *Data_reg* will only be updated when *rd_done* is released. Another triggering signal (*conf* or *rd*) can only be sent after higher-level control module receives feedback signals (*conf_done* or *rd_done*).

TABLE 3.2: LTC2358 control module signals

signals	signal types	description
softspan	input, 24-bit bus	signal from the higher-level module to define the working range of 8 channels
conf	input	signal from the higher-level control module to trigger only conversion
conf_done	output	signal sent to the higher-level control module to indicate the softspan code has been updated
rd	input	signal from the higher-level control module to trigger a normal full two-phase operation
rd_done	output	signal sent to the higher-level control module to indicate a normal two-phase operation has done
data_reg	output, 8 24-bit bus	8-channel conversion results with raw data from adc serial outputs

There are eight LTC2358 chips needed in the analogue interface. Each module has control signals for one chip. Control signals from the higher-level module can be tied together or generated independently depending on the usage scenarios.

3.3.2.3 Switch Chain Controller

Switch chains reconfigure trans-impedance amplifiers and high-speed channels in channels of the memristor crossbar. All switches shown in Figure 3.4 belong to switch chains. The analogue interface uses ADG1414 and ADG1439 to connect switch chains. ADG1414 and ADG1439 have serially controlled eight individual single-pole single-throw switches and two sets of 4-channel multiplexers, respectively. They both use a standard SPI serial interface with signals SCLK, SYNC, DIN and SDO serving as a serial clock, a serial interface enable signal, serial data inputs, and serial data outputs, respectively. A standard transaction transmits 8 bits each time, whereas a daisy chain can transmit any bits of data with every 8-bit data package decoded as control data for one chip.

In the analogue interface, each chain controls one cluster to ensure maximised flexibility and parallelism so that no cross-controlling is needed. A switch chain has 11 chips with 88-bit control data. Each channel uses 11-bit control data, including options for 'channel grounded', 'channel grounded with a capacitor', 'current sources', 'high-speed driver', 'no feedback resistor', '110 kΩ feedback resistor', '820Ω feedback resistor', 'array connected', 'low-side high speed connected', 'high-side high-speed connected' and 'ADC grounded', as shown in Figure 3.4.

The design for a switch chain controller is simple. Triggered by a *wvalid* signal from a higher-level control module such as a read or write controller, the transaction starts by holding the SYNC signal low and shifting one bit into DIN per clock cycle. A switch chain transmits 88-bit data in one transmission under the 25 MHz clock frequency.

3.3.2.4 High-speed Pulse Controller, Arbitrary Logic, Selectors, Current Sources, and Digital Potentiometer

Apart from DAC clusters, ADC clusters and switch chains, there are other parts in the analogue interface to make the control system more versatile. These include a high-speed pulse generator, arbitrary logic, selectors, current sources,

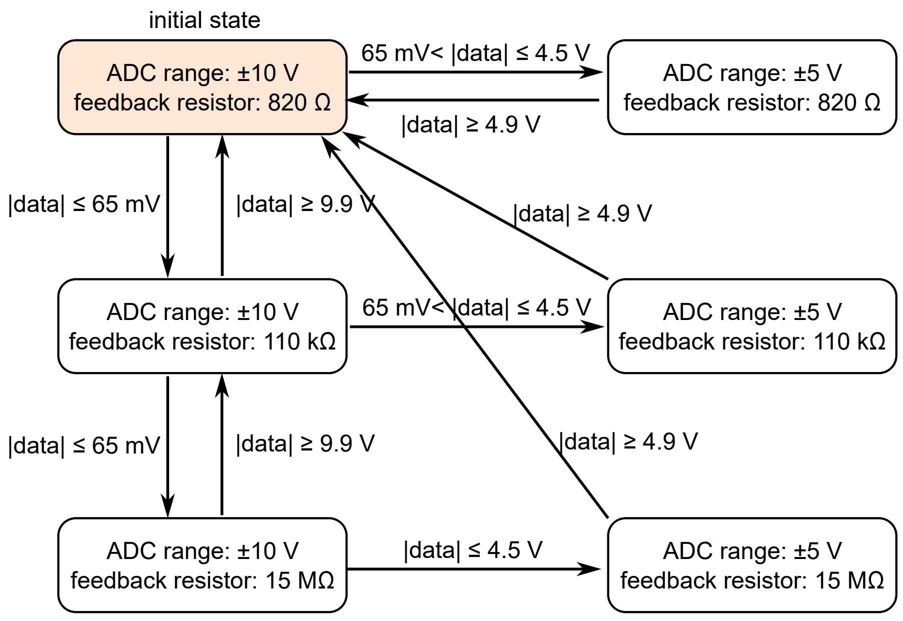

initial state
ADC range: ±10 V feedback resistor: 820 Ω
ADC range: ±5 V feedback resistor: 820 Ω
65 mV< |data| ≤ 4.5 V
|data| ≥ 4.9 V
|data| ≤ 65 mV
|data| ≥ 9.9 V
|data| ≥ 4.9 V
ADC range: ±10 V feedback resistor: 110 kΩ
65 mV< |data| ≤ 4.5 V
ADC range: ±5 V feedback resistor: 110 kΩ
|data| ≤ 65 mV
|data| ≥ 9.9 V
|data| ≥ 4.9 V
ADC range: ±10 V feedback resistor: 15 MΩ
|data| ≤ 4.5 V
ADC range: ±5 V feedback resistor: 15 MΩ

FIGURE 3.10: Auto-range module state machine

and a digital potentiometer.

High-speed channels are channels controlled by MOSFETs to generate short pulses. In the digital interface, they are directly generated via the user I/Os on FPGA with a timer to control the pulse width according to the pulse width parameter passed from the transmission layer. Arbitrary logic uses an ADG1414 switch chip, one of the same chips used in a switch chain, to flexibly select references for current sources, selectors, etc. by users. It can simply reuse a control module for a switch chain with only 8-bit data. Selectors and current sources use a single ADG1414/ADG1439 chip. Therefore, they can reuse the switch chain controller module without a daisy chain. Digital potentiometer module is to control the digital potentiometer chip AD5293BRUZ-20 [178]. This chip uses SPI protocol to communicate, so the control module is similar to the DAC controller module.

3.3.2.5 Auto-range Module

One of the memristor crossbar control circuits' main functions is to read back currents in crossbar bitlines. Trans-impedance amplifiers convert current to voltages to use ADCs to measure current. However, memristor resistance states span from kohms to Mohms, making it hard to measure the voltage accurately with only a single fixed-value feedback resistor in a trans-impedance amplifier. In each channel, the analogue interface has three feedback resistors with different values - 820 Ω, 110 kΩ, and 15 MΩ - so a proper feedback resistor can be chosen in the current read back. Also, in LTC2358, analogue input ranges can be altered with different softspan codes for more accurate conversion results. Therefore, the digital interface includes a module to implement the algorithm that can automatically select the best feedback resistor and ADC input range according to read-out voltage results.

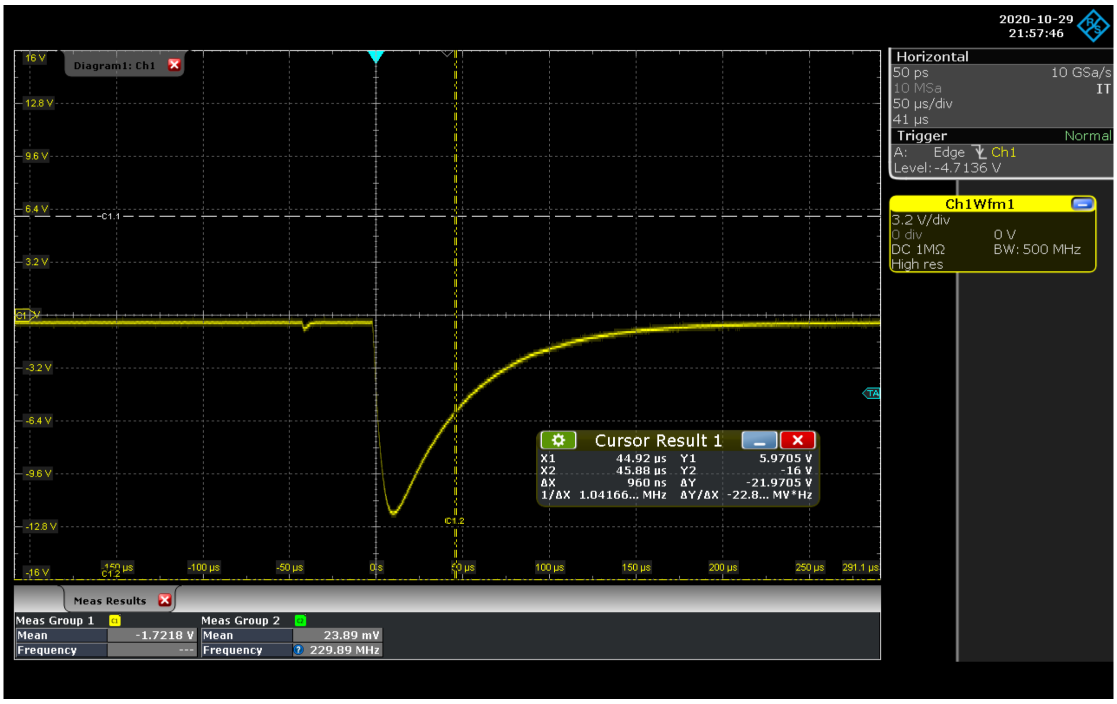
Figure 3.10 shows the finite state machine of the auto-range module. A different choice of feedback resistor can make over 100x differences, whereas ADC input range change only causes 2x differences in conversion results. Therefore, choosing a proper feedback resistor should be the first step in every state transition. Also, as a transition condition, conversion voltage thresholds should have certain margins to avoid misjudgement due to possible inaccurate conversion results. Thus, thresholds were chosen as transition conditions are ± 4.5 V and ± 65 mV for ADC ranges and feedback resistors. The start point always uses the maximum possible ADC range (± 10 V) and the smallest feedback resistor (820 ohms) to avoid voltage overflow. The state machine automatically selects a larger feedback resistor if the conversion result is within the range of ± 65 mV. When the feedback resistor selection process finishes, the conversion result compares with ± 4.5 V to choose a better ADC input range. Once both feedback resistor and ADC range are the best options, another conversion starts to check if saturation happens with the new range and new feedback resistor. If yes, the state machine changes back to the start point, and the auto-range starts again to find the best configuration. This state will not end until the best settings are found without saturating ADCs.

Several control signals are used between the auto-range module and its higher-level control module. They are listed in table 3.3. Signal *auto_range_en* is used to enable auto-range, and it will be turned off once auto-range finishes. Signal *channel_active* is used to turn on the channel that runs an auto-range operation. *Data_reg* passes 18-bit ADC conversion result to the auto-range module, *softspan*, *switch*, and *auto_state* give the updated softspan and switch configuration as well as the auto-range state in this state. When the new ADC range and switches data are applied from the higher-level module, another measurement starts, and the auto-range module gives the new state according to the measured result. If the best state has been found, the state stays, and signal *auto_range_done* sets high to indicate the auto-range has finished.

Auto-range is automatically run in a basic read operation. Auto-range is one of the states in the read operation state machine. After this module successfully

signals	signal type	description
auto_range_en	input	signal from higher-level control module to enable autorange
channel_active	input	signal from higher-level control module to turn on the channel for autorange
data_reg	input, 18-bit data bus	adc conversion data sent to run auto range
softspan	output, 3-bit	updated softspan to change adc range
switch	output, 2-bit	updated switch configuration to change feedback resistor
auto_state	output, 6-bit one hot code	auto range state
auto_range_done	output	signal sent to higher-level module to indicate the final best state has been found

TABLE 3.3: Signals in auto range module



```
Horizontal
50 ps                10 GSa/s
10 MSa                     IT
50 µs/div
41 µs
Trigger                Normal
A:     Edge  Ch1
Level: -4.7136 V

Ch1Wfm1
3.2 V/div
0 div          0 V
DC 1MΩ         BW: 500 MHz
High res

Cursor Result 1
X1       44.92 µs   Y1       5.9705 V
X2       45.88 µs   Y2          -16 V
ΔX         960 ns   ΔY     -21.9705 V
1/ΔX 1.04166... MHz ΔY/ΔX -22.8... MV*Hz

Meas Results
Meas Group 1  C1            Meas Group 2  C2
Mean        -1.7218 V       Mean          23.89 mV
Frequency         ---       Frequency ? 229.89 MHz
```

FIGURE 3.11: Illustration of the large spike when changing states. When the feedback resistor changes, a voltage spike up to +/- 12.5 V can appear in output terminals of trans-impedance amplifiers. This is because the voltages at the output terminals cannot change immediately.

picks the best combination of the feedback resistor and ADC range, the configuration will stay unchanged until the next round of auto-range. More details will be shown in a later subsection for read operation controller. Auto range function not only gets a more accurate current with the best resolution but also can be used to derive memristor resistance states which can be important information for testing and debugging.

3.3.2.6 Dynamic Comparison Module

According to experiments, when the feedback resistor changes from 110 Kohms to 15 Mohms, a voltage spike up to +/- 12.5 V appears in output terminals of trans-impedance amplifiers. This is because the voltages at the output terminals cannot change immediately. Figure 3.11 shows the spike we observed from an oscilloscope. If ADCs sample the voltage before settling in the auto-range process, the state machine may stay in an error state. Therefore, to guarantee the fluctuation is within an acceptable range, the digital interface has a module with an algorithm that can dynamically compare two successive measured voltage values with the sampling frequency of 50MHz. Experimental results show that if the voltage difference between two measurements is smaller than 0.04V, we assume that the voltage has settled because 0.04V is smaller than the 65mv threshold for the transition in the auto-range state machine. In the design, the dynamic comparison module takes two successive measurements from the same channel, and the auto range will only be enabled when the voltage meets the requirement.

3.3.2.7 Read Operation Controller

For a basic read operation, the working scheme is as follows: firstly, set DACs in wordlines to V_{bias} and in bitlines to 0 for a selected device, and $V_{bias}/2$ in both wordlines and bitlines for unselected devices. Meanwhile, initialise switch chains to enable trans-impedance amplifiers, and set selectors to drive transistors for using a selector-based array. After waiting for DACs output voltages to settle and switches to perform successfully, the auto-range process starts to find the best ADCs range and feedback resistor by firstly configuring ADC softspan. Next, run conversion in ADCs and load ADC results. If this conversion is not the first one after changing the auto-range state, the conversion results will be sent to the 'dynamic comparison module' to be compared with the last result, which has the same ADC range and feedback resistor. If the voltage difference is smaller than 40 mV which is smaller than the auto-range threshold of 65 mV, the voltage fluctuation is small enough to go to a correct auto-range state. After that, the auto-range process is enabled for this conversion result. If *auto_range_en* signal from the 'auto-range module' is high, it indicates that the best auto-range state is found. Now ADCs can start loading ADC conversion data while keeping the current configuration. ADCs sample 32 times in a read operation to get average values as final results. A read operation in an actual

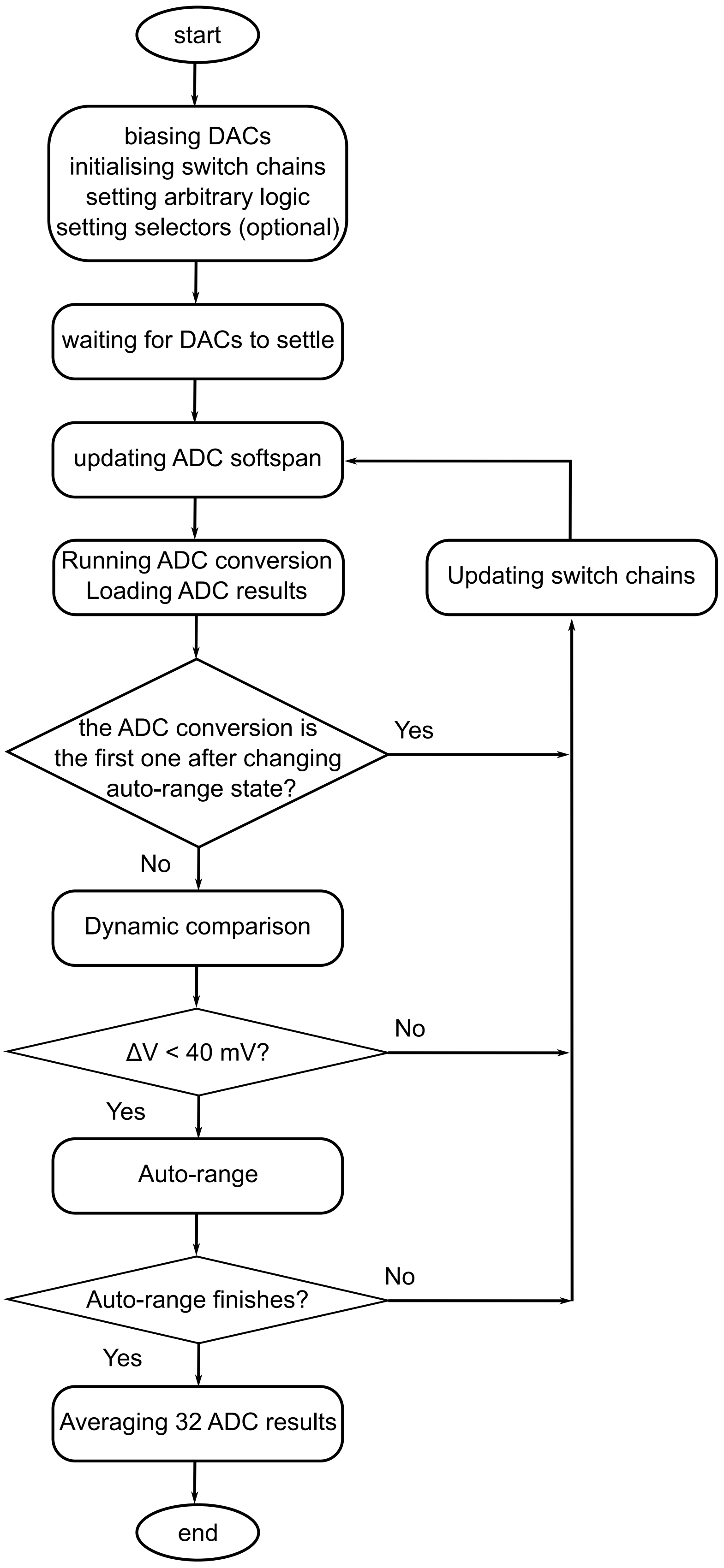


FIGURE 3.12: Flowchart of the read operation

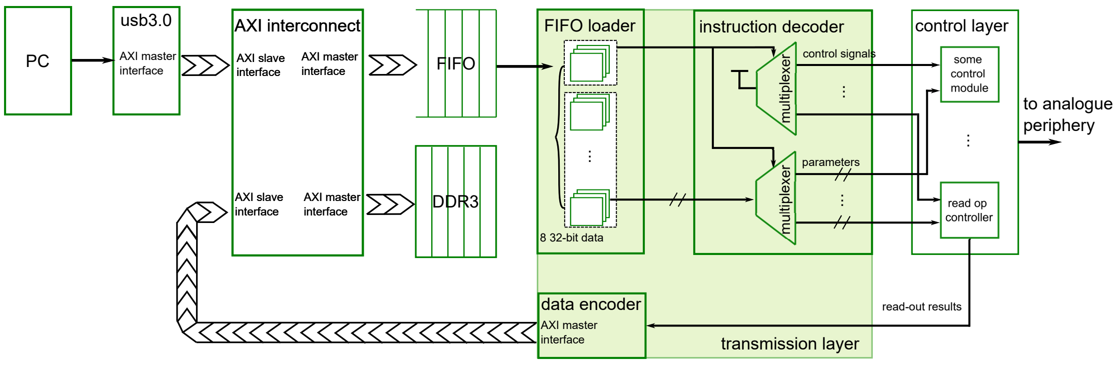


FIGURE 3.13: Transmission layer

practice usually starts after a write operation to verify memristor conductance. The flowchart for a read operation is shown in Figure 3.12.

3.3.3 Transmission Layer

The transmission layer receives instructions from a PC, decodes the instructions, sets the control signals for the control layer, packs the read-out results, and sends the results to DDR3, as shown in Figure 3.6. The transmission layer consists of a FIFO loader, an instruction decoder, and a data encoder, as illustrated in Figure 3.13.

When a PC sends an instruction to the FPGA, the instruction is delivered to the FIFO via the USB3.0 IP. We design an instruction set with each instruction using eight 32-bit data. Please refer to Appendix A, section 2, to see the instruction set details. The FIFO loader loads the eight 32-bit data each time and sends the full instruction package to the instruction decoder. When the execution of the current instruction finishes, the FIFO loader loads another instruction package unless the FIFO is empty.

The instruction decoder is used to decode instructions. Each instruction contains an opcode with specialised parameters designed for this specific instruction. By checking the opcode, the instruction decoder sends the extracted parameters from the instruction and sets the control signals to enable lower-level modules with the aid of the pre-defined instruction sets.

When a read operation is enabled, the read-out results are fetched from the control layer to the data encoder. The data encoder packs the read-out results and sends them to DDR3 memory with a ready flag sent to another specified address. When the software interface in the PC detects a valid ready flag, a read request from the PC can be issued, and the read-out results can be loaded from the DDR3 memory to the PC.

In this work, inter-IP communication is delivered via Advanced Microcontroller Bus Architecture (AMBA) AXI interface [179]. AMBA AXI interface is a burst-based transaction interface with independent read address, read data, write address, write data, and write response channels. The address channels contain the control information defining the transaction, and the data channels contain transferred data between the master interface and the slave interface. In a read transaction, the master interface sends a read request in the read address channel to the slave interface, and the slave interface transfers data to the master interface through the read data channel as a response. In a write transaction, the control information and data are both transferred from the master interface to the slave interface, and the write response channel is used to signal the completion of a transaction from the slave interface to the master face. AXI slave interfaces are integrated into the FIFO and the DDR3 controller IPs generated by the Vivado toolkit, and AXI master interfaces are involved in the USB3.0 IP provided by Cesys and the data encoder in the transmission layer. An AXI Interconnect IP [180] is used to interconnect between two master interfaces and two slave interfaces.

3.3.4 Clock Domain Crossing Synchronisation

More than one clock is used in this work to meet the speed requirement in different modules. More specifically, the USB3.0 IP, the DDR3 controller IP, and the AXI interface are running with a 100MHz clock. The ADC module and the DAC module use a 50MHz clock. The switch chain controller uses 25MHz. Without clock crossing domain synchronisation, the setup/hold timing requirement might not be met, and metastability might occur [181]. Therefore, clock domain crossing synchronisation is implemented to ensure steady data transmission across different clock domains.

A simple handshake is implemented for multi-bit data transmission across clock domains: when the circuit in the sending domain has the data ready, it sends a request signal to the data receiving domain and holds it until it has been successfully received. In the receiving domain, a two-stage flip-flop is used to remove the metastability. When the circuit in the receiving domain successfully receives a valid request, it sends back a ready signal to the sending domain and passes the multi-bit data to two-stage flip-flops.

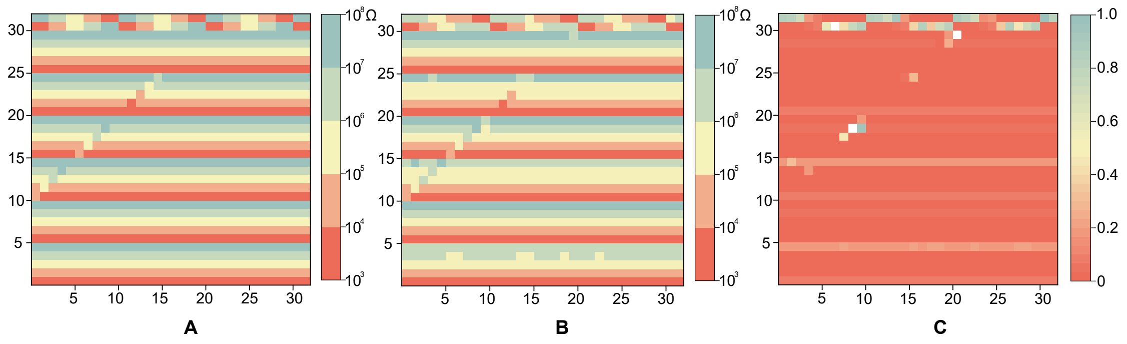


FIGURE 3.14: Array read results for a 32×32 resistor array. **(A)** Resistor array used in the experiment with resistors ranging from 1kΩ to 15MΩ. **(B)** Measured resistance. **(C)** Proportional read-out errors.

3.4 Experiments

In this section, two commonly used functions, array read and increment pulsing programming, are validated through two experiments. These experiments will show how the digital interface can handle devices and deliver advanced functions in testing and characterisation.

3.4.1 Resistor Array handling

The objectives of the control and characterisation system built with the FPGA-based digital interface can be summarised as three points: device characterisation, device handling and higher-level function mapping. For device characterisation and handling, parallel read/write operations improve the time efficiency in testing and controlling; for higher-level function mapping, parallel read/write operations accelerate the matrix multiplication. In this subsection, a resistor array handling experiment is illustrated to validate the system capability of conducting efficient and accurate crossbar read operations.

Figure 3.14 shows the array read results for a 32×32 resistor array. The used resistor array provides known resistance values to calculate the read errors. Figure 3.14 **(A)** gives the actual resistance of the array. The resistance ranges from 1kΩ to 15MΩ. Line-parallel read operations were performed on each row to collect read-out results (shown in Figure 3.14 **(B)**), and the proportional errors are calculated as $|R_{measured} - R_{actual}|/R_{actual}$ (shown in Figure 3.14 **(C)**). The read-out performance is excellent, with 802 out of 1024 resistors having less than 5% measured errors. The exceptional resistors are placed intentionally. The system

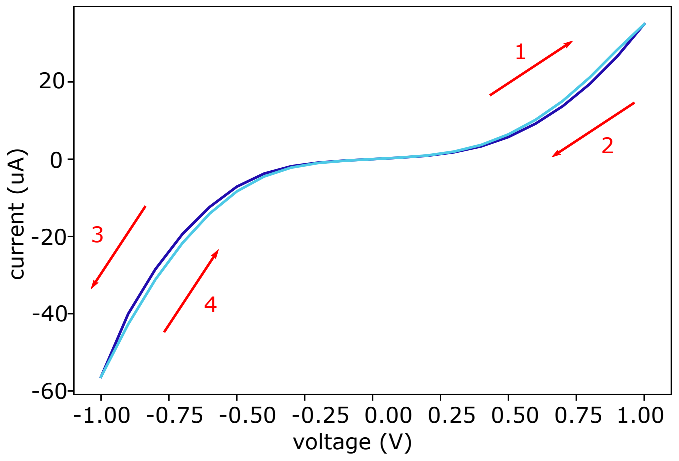


FIGURE 3.15: IV sweep on a Pt (12 nm)/ TiOx (25 nm)/ AlOx (4 nm)/ Pt (15 nm) memristor device. The bias voltage sweep sequence: 0V → 1.0V → 0V → -1.0V → 0V.

selected the feedback resistors suitable during auto-range for those 'normal' resistors but not for those exceptional ones. Therefore, the unsuitable feedback resistors caused higher errors in those exceptional resistors.

3.4.2 Memristor Current-Voltage Sweep

Increment pulsing programming is another commonly used function for a memristor control and characterisation system to explore the device characteristics. In this subsection, an experiment of IV-sweep for a Pt/TiOx/AlOx/Pt device [64] is presented to validate the capability of the digital interface for conducting increment pulsing programming.

Figure 3.15 shows the IV characteristic curve of the tested device. To conduct the experiment, the bias voltage swept from 0V → 1.0V → 0V → -1.0V → 0V with an increment step of 0.1V. The memristive resistance states were measured, and the current flowing through the device was calculated and plotted in the curve. The curve shows IV hysteresis for bipolar devices [182] that switch from a low resistive state (LRS) to a high resistive state (HRS) and then back to LRS. Due to the very low bias voltages and the sensitivity of the switching, the contrast between the HRS and the LRS is relatively small. However, the goal of this experiment is to validate the instrument rather than prove how obvious the switching is in this device. This experiment proved the digital interface capability of conducting increment pulsing programming for real memristor devices.

3.5 Discussion

This chapter has presented an FPGA-based digital interface for memristor arrays. Whilst this was explicitly designed for memristors, it can also be involved in building a much more powerful, versatile tool for general-purpose characterisation. This characterisation instrument allows simultaneous read/write operations to memristor arrays, making it a helpful hardware infrastructure that can not only accelerate the testing but also be employed as a part of the board-level memristor-based neuromorphic systems. To be more specific, When writing devices, parallel write allows sending different bias voltages in different channels to write devices in the same bitline at the same time. Without parallel write, this procedure takes N times with N as the number of devices. Similarly, when doing a basic read operation for MAC operations, parallel read allows biasing devices in the same bitline at the same to make sure the measured current is the accumulated current through all branches. Parallel read and write ensure the functionality and efficiency of MAC operations.

TABLE 3.4: Comparison between the instrument built with this work and similar systems. Data shown in this table has been published in the author's work [169]

	[183]	[184]	[185]	This work
Parallel read	No	No	No	Yes
Parallel write	No	No	No	Yes
Channel count	2R+2W+16D	32R+32W	4R+2W	64R/W+64D
Form factor	Portable	Desktop	Benchtop	Desktop
Min. chan. current	N/A	1nA	10nA	100nA
Max. chan. current	N/A	5mA	500mA	12mA
Current sample rate	N/A	50-1000Ss^{-1}	N/A	833Ss^{-1}
Voltage resolution	166/665μ V	3/24mV	1 μ V	78 μ V
Voltage sample rate	100MSs^{-1}	200kSs^{-1}	1.25GSs^{-1}	100kSs^{-1}
Min. pulse width	N/A	90ns	10ns	40ns
Max. chan. current	N/A	±5mA	±500mA	±12mA
Pulse volt. range	±5V	±12V	±20V	±13.5V
Power	500mW	4.5W	2W	20W

The instrument built with the digital interface presented in this chapter can achieve competitive performance compared to existing commercially available testing instruments. The comparison between the instrument built with this work and other similar systems is shown in Table 3.4. The results have demonstrated that the parallelism of the digital interface presented in this chapter facilitates the high data throughput testing for emerging memory devices, though with a relatively low sampling rate. Thus, we foresee that the control system built with the digital interface will play a critical role in accelerating research regarding memristor-based neuromorphic designs where parallelism and high-speed data acquisition are of paramount importance.

Chapter 4

NeuroPack: A Software Simulator for Memristor-based SNNs

4.1 Introduction

Neuromorphic computing has witnessed immense growth in the last decade, manifested in advances, in theory, hardware, and infrastructure. Neuromorphic computing has proposed a wide range of artificial neural networks [156] configurations, such as LSTMs [186] and convolutional neural networks (CNNs) with different weights and signal quantisation [187], spanning both spiking [188] and non-spiking approaches [189]. There is no doubt that this design process entails multiple decision points, which results in a vast and complex design space. Meanwhile, neuromorphic hardware is increasingly employing memristors thanks to their multibit storage capability [64] and their simple architecture that can be tessellated in large arrays [190]. Along these lines, software-based simulation platforms designed for memristor-based neuromorphic systems become significant for the fast validation of design ideas and predicting device behaviour.

Current simulators (e.g. MNSIM [82] and NeuroSim [81]) centre more on circuit-level designs, serving as tools either to simulate the behaviour of different hardware modules or to estimate the performance of memristor-based neuromorphic hardware in integrated circuit designs. Sitting at a higher level of abstraction would be an ′algorithm-level device-model-in-the-loop′ simulator (or ′algo-simulator′ for short) designed to test functionally defined (as opposed to explicitly designed) circuits with memristive device models at the algorithm level, e.g. performing specific online or offline learning tasks with memristors as synapses in spike-based NNs. Such a tool would allow fast verification of design concepts before serious hardware design effort is committed. It essentially answers the question: Is my design likely to function given knowledge of my memristive devices, assuming the rest of the circuit functions flawlessly? If yes, work can proceed to the next stage.

This chapter presents NeuroPack: a simulator incorporating memristor models for neuromorphic computing at the algorithm level. NeuroPack is a complete, hierarchical framework for simulating spiking-based neural networks, supporting various neuron models, learning rules, memristor models, memristor devices, neural networks, and different applications. Written in Python, it can be easily extended and customised by users. NeuroPack also integrates an empirical memristor model proposed by [88]. Between processing algorithms and setting & monitoring memristor states, there is the significant step of applying a pulse of specific voltage and duration to trigger a memristor state change corresponding to some desired weight change calculated from learning rules. NeuroPack integrates a module to convert desired weight changes to estimated stimulation pulse parameters for bridging this gap. Besides, NeuroPack can also connect with commercially available instruments such as ArC 1 [191] to use parameters extracted from real devices. In terms of applications, we use NeuroPack to demonstrate image classification on the MNIST dataset [192] in our 'Results' section. We also give result analysis for systems with different values of the **R tolerance**, a parameter used in weight updating, and two biasing methods as examples to showcase that NeuroPack assists users in investigating how critical design, device and architectural factors affect memristor-based neuromorphic computing systems. Finally, NeuroPack includes a built-in analysis tool with a user-friendly graphic user interface (GUI) for visualising and processing classification results. The main contributions of this work include:

1. Developing an algo-simulator for memristor-powered neuromorphic computing with selectable neuron models, device models, and learning rules.
2. Modelling memristor state changes in neuromorphic computing tasks given user-defined memristor parameters.
3. Converting expected weight changes prescribed by learning rules into parameters of bias pulses used for triggering memristor state changes in weight updating.

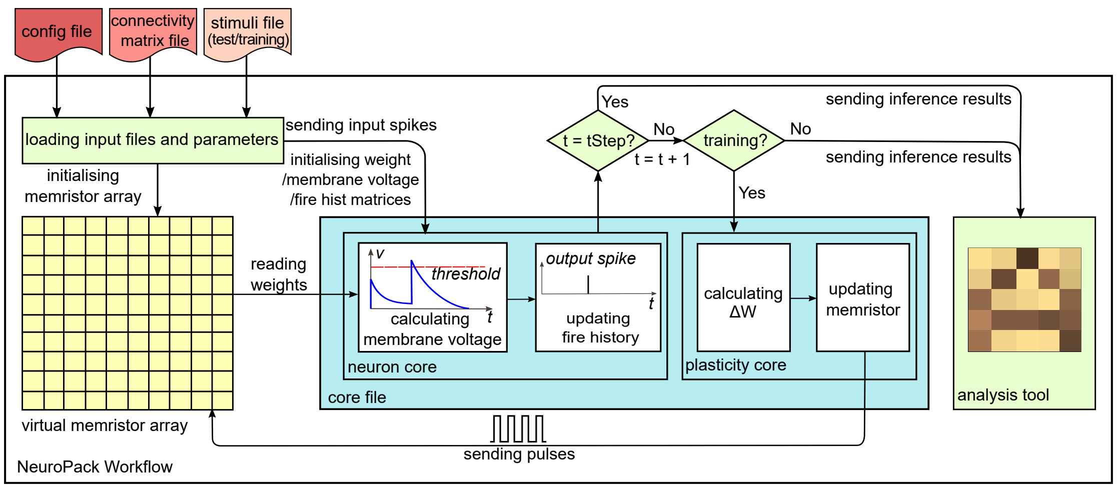


FIGURE 4.1: NeuroPack workflow. NeuroPack takes three input configuration files: the connectivity matrix, the (neural) stimuli file, and the config file. The virtual memristor array is formed by memristor models with several memristor devices initialised according to instructions in the configuration and connectivity matrix files. Neuron and plasticity models are included in the core file. There are three steps in the 'updating fire history' stage: calculating fire states assuming neurons fire 'freely', adding network-level constraints, and updating firing history. Most learning rules except STDP support 'Calculating ΔW'. Users can easily employ different neuron and plasticity models by replacing the core file. The image is from the author's publication [193].

The rest of the chapter is organised as follows: Section 2 introduces the architecture of NeuroPack with core parts and the workflow. An application of handwritten digit recognition in the MNIST dataset is demonstrated in Section 3, and the paper is summarised in Section 4.

This work has led to these publications [193, 194].

The source code of the work presented in this chapter can be found in the Github repository: https://github.com/hjq310/NeuroPack

4.2 Methods

4.2.1 Design Overview

NeuroPack is designed to predict the results of online learning or offline classification tasks with selectable neuron, plasticity and memristive device models and to monitor memristor state changes. NeuroPack's workflow (see Figure 4.1) is divided into five parts to cover the functionality of achieving those two tasks: input file handling, virtual memristor array, neuron core, plasticity core, and analysis tool.

Three input files need to be provided from users for input data handling: *configuration file*, *connectivity matrix file*, and *stimuli file*. The *configuration file* includes the core parameters for setting up NN topology (e.g. neuron number in each layer, number of layers etc.), neuron models (e.g. leakage term, read noise scale etc.), and memristor devices (e.g. upper and lower boundaries of memristor resistance). The *connectivity matrix file* defines neuron connectivity and maps synapses to virtual memristors. The *stimuli file* contains input samples and labels. Notably, NeuroPack only receives inputs as spike trains. Before feeding the input to the simulator, the raw (sensor) input should be encoded independently into spikes. User input files can be loaded through the NeuroPack main panel, whose details can be found in Appendix B, section 4.

We now go through the steps for executing a classification task in a memristor-based SNN. Firstly, the neuron model input samples are transformed into spikes and stored in the *stimuli file*. The training and test datasets are loaded independently. Because NeuroPack employs a framework of non-overlapping consecutive abstract time intervals at each time step, an input neuron is firing (1) or non-firing (0) for precisely one time step. After that, the neuron core reads the weights stored in the virtual memristor array and updates internal variables, such as membrane voltages, based on the chosen neuron model. The new firing states are then computed in two steps: first, determining whether neurons should fire by checking if the membrane voltages are above the threshold, and second, adding network-level inhibition (e.g. winner-take-all networks). When the training is enabled, inference results are passed to the plasticity core to update weights using the selected learning rule; otherwise, plasticity updates are skipped.

Weights are updated in the plasticity phase. For local timing-dependent learning rules such as STDP, long-term potentiation (LTP) or long-term depression (LTD) are applied to memristor devices directly using spike timing information. For non-local gradient-based learning rules, output errors are back-propagated. Extra information such as input labels available to the system *configuration file* is needed to calculate the output errors. Notably, there are two major conceptual approaches to describing plasticity events. The more straightforward way is to define a function that transforms plasticity-related variables directly into pulsing parameters. With the assistance of this function, the memristor switching dynamics will determine the actual resistive state (RS) change, which in turn will be converted to a weight change. The more complex route transfers plasticity-related variable configurations to a specified weight change and then searches the device model (a model is required for this technique) for a solution predicting that some collection of pulsing parameters will result in the required change. Processing every sample requires repeating the same process. After inference results are acquired, they can be sent to the built-in analysis tool together with internal variables and parameters, including synaptic weights, membrane voltages and firing history, for data visualisation and analysis.

4.2.2 Neuron Models and Learning Rules

Neuron and plasticity rules models are located in the ′neuron core′ and the ′plasticity core′, respectively, as shown in Figure 4.1. The gradient calculation depends on the neuron model in gradient-based non-local learning rules, such as backpropagation. A straightforward way to implement this dependency employed in NeuroPack is to pair each learning rule with one neuron model and place them together in the same ′core′ file to allow the strong interdependences between the neuron model and the learning rules. NeuroPack offers four example core files with different neuron and plasticity model pairs: the rate-coded leaky integrate-and-fire neuron (LIF)[89] with STDP, the rate-coded LIF with backpropagation (BP)[107], the temporal-coded LIF with Tempotron [195], and Izhikevich neuron [105] with direct random target projection (DRTP) [196]. Users can introduce their core files to include other neuron and plasticity rules according to their needs by extending existing example cores as standard templates. In this section, we use one specific example, the core file with the rate-coded LIF neuron model and BP learning rule pair, to illustrate how NeuroPack constructs a fully-connected multi-layer spiking neural network with winner-take-all functionality [90].

4.2.2.1 Leaky integrate-and-fire neuron and Back-propagation

The leaky integrate-and-fire neuron model is one of the simplest and most computationally friendly neuron models that can still benefit from the biological plausibility of SNNs. Most neuromorphic accelerators use LIF neurons, regardless if memristors are involved (e.g. [72]) or not (e.g. [53, 126]). NeuroPack implements LIF with the equations shown below adapted from the differential form [197] by assuming discretised time steps:

$$\begin{aligned} V_t &= \sum W x_t + \alpha V_{t-1}(1 - y_{t-1}) \\ y_t &= h(V_t - V_{th}) \end{aligned} \tag{4.1}$$

Where V_t denotes the membrane voltage at time step t, W represents the weights, x_t gives input spike to the neuron (represented as 1 or 0), $\alpha \in [0, 1]$ denotes a leakage term, y_t is the output spike at time step t, $h(x)$ is the Heaviside step function, and V_{th} is the threshold. The equations illustrate that two parts determine the neuron membrane voltage V_t at any time step: the current response calculated by the weighted sum of incoming spikes at time step t, and the history membrane voltage at time step $t-1$. An output spike is released if the membrane voltage surpasses its threshold, and then the membrane voltage is reset to 0. The mean-square-error (MSE) cost function E at time step t is shown as follows:

$$E(t) = \frac{1}{2N}\sum_{i=0}^{N}(y_{i,t} - \hat{y}_{i,t})^2 \tag{4.2}$$

where N is output neurons numbers with i denoting the output neuron index, and $\hat{y}_{i,t}$ gives the correct firing state of output neuron i at time step t. Using the backpropagation with the chain rule (see derivation details in Appendix B, section 1), the final expression of the weight changes is shown below:

$$\begin{gathered} \Delta W_k = -\eta \delta_{k,t} x_{k,t}^T \\ \delta_{k,t} = \begin{cases} \frac{1}{N}(y_{k,t} - \hat{y}_t) \odot h'(V_{k,t} - V_{th}) & \text{if k = K} \\ (W_{k+1,t}^T \delta_{k+1,t}) \odot h'(V_{k,t} - V_{th}) & \text{otherwise} \end{cases} \end{gathered} \tag{4.3}$$

Where η, k, and K are the learning rate, the layer index, and the total number of layers, respectively. W_k gives the weight matrix between layer $k-1$ and k. $\odot$ represents element-wise multiplication. δ gives the error back-propagated from the output neurons. Notably, gradient descent cannot directly be implemented in SNNs, because of the non-differentiability of the Heaviside step function $h(V_t - V_{th})$. Using surrogate gradient descent [198] with straight through estimators (STEs) [199, 200] is a common solution to address this issue. NeuroPack applies a surrogate derivative proposed by [201]. Please refer to Appendix B, section 1 for the complete derivation. The variable $\hat{y}_{i,t}$ is included in *stimuli file* as labels of input samples, and everything else is accessible directly to NeuroPack.

4.2.2.2 Adding winner-take-all functionality

This subsection introduces how NeuroPack adds winner-take-all (WTA) functionality to constrain at most one firing neuron in the output layer at each time step. Adding WTA functionality is done by adding an extra softmax layer and only allowing the neuron with the largest firing probability to fire:

$$S_t = softmax(V_t \odot y_t) \tag{4.4}$$

The cost function is changed to the cross-entropy form correspondingly:

$$E = -\sum_{i=0}^{N} \hat{y_{i,t}} ln(S_{i,t}) = -ln(S_{j,t}) \tag{4.5}$$

Where j is the index of the output neuron that should fire. By calculating the gradient of the new cost function, weight changes can be expressed as:

$$\Delta W_k = -\eta \delta_{k,t} x_{k,t}^T$$
$$\delta_{k,t} = \begin{cases} (S_t - \hat{y}_t) \odot (y_t + V_t \odot h'(V_{k,t} - V_{th})) & \text{if k = K} \\ (W_{k+1,t}^T \delta_{k+1,t}) \odot h'(V_{k,t} - V_{th}) & \text{otherwise} \end{cases} \tag{4.6}$$

All variables are accounted for before the WTA application; the freshly introduced softmax function is placed in the same core file by adding network-level constraints. For the complete derivation, please refer to Appendix B, section 2.

In summary, NeuroPack takes parameters such as learning rate, read noise scale, the threshold and the leakage from the *configuration file* to set up a multi-layer spiking neural network. Input spikes are fetched from the stimuli files and delivered to the 'neuron core' in the core file. The 'neuron core' then reads the memristor RSs from the virtual memristor array, computes membrane voltages, and updates firing states during the inference phase. If training is enabled, inference results, the correct firing states, and internal variables are loaded into the 'plasticity core' to calculate weight changes which are later used to trigger memristor RS updates accordingly, as illustrated in Figure 4.1.

4.2.3 Memristor Model

In the virtual memristive device shown in Figure 4.1, a data-driven device model is involved in predicting memristor switching dynamics as a function of the bias voltage and the current RS responding to plasticity events. Whilst this emulation platform has been developed to interface with any voltage-driven memristor model from the literature, throughout this work, Neuropack was validated with the model of [88]. This model captures how the new memristive resistance states depend on the current states and the bias voltages, and it is an empirical one that matches well experimental results acquired by technologies developed in house [202]. This model can be flexibly configured to model a wide range of memristor devices by providing different values of parameters. The model describes memristive RS switching rate (dR/dt) as a function of the initial RS and the bias voltage, as reproduced here for convenience:

$$\frac{dR}{dt} = m(R, v) = \begin{cases} A_p(-1 + \mathbf{exp}(|v|/t_p))(r_p(v) - R)^2 & \text{if } v > 0, R < r_p(v) \\ A_n(-1 + \mathbf{exp}(|v|/t_n))(R - r_n(v))^2 & \text{if } v \leq 0, R \geq r_n(v) \end{cases} \tag{4.7}$$

where A_n and A_p are scaling factors. The $(\mathbf{exp}(|v|/t_{p,n}) - 1)$ term describes the main, exponential dependency of the switching rate on the bias voltage with fitting parameters t_n, t_p retrieved from modelling. The $(r_{p,n} - R)^2$ term encapsulates the 'RS saturation' of the switching rate on the current RS with the aid of fitting parameters a_{0p}, a_{1p}, a_{0n}, a_{1n}: the closer the RS is to the upper (lower) limit, the harder it is to further push it up (down). Fitting parameters are used in $r_{p,n}(v)$, a simple, first-order polynomial helper function that captures the nature of the switching rate's dependency on the current RS:

$$r(v) = \begin{cases} r_p(v) = a_{0p} + a_{1p}v & \text{if } v > 0 \\ r_n(v) = a_{0n} + a_{1n}v & \text{if } v \leq 0 \end{cases} \tag{4.8}$$

Despite various forms of models (e.g. charge-flux models [203]), the essential condition observed is that the device model always takes time-wise discretised voltage series and then calculates the current response and the RS change. NeuroPack does not support models that take current as inputs presently. dt is a global, pre-defined parameter used throughout the simulation of memristor resistance updates. d can be loaded from the configuration file. Memristor models are organised in the virtual array in NeuroPack by using a class **ParametricDevice(*args)** that takes user-defined parameters as inputs to create a device object. Methods **initialise(R)** and **step_dt(V, dt)** in the same class are used to initialise the memristor device with specific resistance and update the memristor RS by applying a pulse with magnitude of V and pulse-width of dt correspondingly. The 'virtual memristor array' class also includes methods to read the device RS with the wordline **w** and the bitline **b** (**read(w,b)**), and to apply a pulse with magnitude of **v** and pulse-width **pw** to the device at said address correspondingly (**pulse(w, b, v, pw)**). When we apply a 'write' operation, the **pulse(w, b, v, pw)** method is called. The **pulse(w, b, v, pw)** method is called for a ’write’ operation, which takes **pw** to split it into multiple successive sub-pulses with the duration of *dt* to be applied to memristors. For a ’read’ operation, by default, NeuroPack ignores the effect of the read-out voltages on memristor RSs to reduce simulation runtime, but it provides an open option for users to determine whether to consider the read-out voltages. It is these functions that the core file calls to access and update memristor RS, required by the neuron models and learning rules. By changing model parameter sets corresponding to different types of devices with different stacks, technologies, or even underlying electrochemical mechanisms, NeuorPack can serve as a great tool to understand how different memristive devices can influence the training and the inference of basic machine learning algorithms. The Memristor model mainly captures the non-linear switching dynamics, and the read variations are added in method **read(w, b)**.

4.2.4 Weight Mapping and Updating

The basic operating principle of most memristor-based neuromorphic designs (e.g. [57] and [204]) is analogue signal processing to accelerate matrix multiplications. By mapping synaptic weights to memristor conductance and applying voltage pulses to wordlines of the memristor arrays, the weighted sum of the inputs is represented as the current in the bitlines, according to Ohms’ law. To do so, Neuropack maps weights linearly to memristor conductance to restrict weight within range [0, 1] by default, but other mapping methods can also be chosen if needed. The linear mapping represents weights using a mapping function: $W = \alpha G + \beta$. Here, α and β are coefficient and offset. Negative weights are not used because they can be simply converted to positive weights with different α and β. Hence, differential pair mapping is not applied by default. However, achieving well-controlled RS changes in nonlinear devices like memristors is challenging because the resulting RS depends on both the current RS and pulsing parameters (typ. magnitude and pulse-width). To address this issue, a module for predicting the expected pulsing parameters that can produce the desired memristor RS changes is included in NeuroPack. This module uses memristor model parameters for creating a virtual memristor device object, takes the current memristive RS, the target RS, a user-defined time step dt, and a list of different pulsing parameter sets as inputs, and returns the best pulsing parameters in the options list that can produce the memristor RS closest to the target.

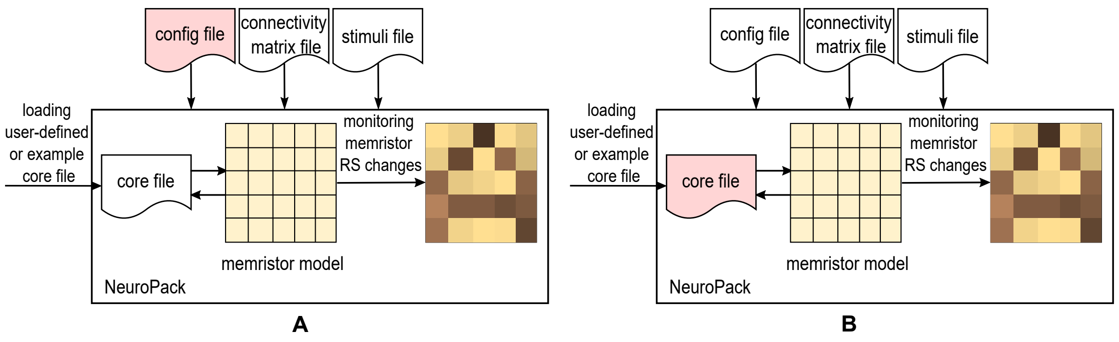


FIGURE 4.2: Two usage scenarios of NeuroPack. Different targeted parts in two scenarios are coloured in pink. (**A**) In the first scenario, users can explore the system sensitivity to a global parameter defined in the 'config file'. (**B**) In the second scenario, users can verify the system-level concepts by changing neuron and plasticity models. The image is from the author's publication [193].

Inside the module, firstly, a **ParametericDevice** object is created using user-provided memristor model parameters. Next, the **step_dt(V, dt)** method is called to compute all resulting resistance by taking the pulsing parameters in the options list. After that, the module calculates the differences between the resulting resistance and the target one and returns the parameter set with the minimum difference. Finally, the pulse with the chosen parameters is applied to the virtual memristor device. The pseudocode of this module can be found in Appendix B, section 3.

The complete weight update procedure taking 'prediction-write-verify' loops with the use of the pulse parameter selection module is as follows: to begin with, two stop conditions are set by taking two user-defined parameters, the **R tolerance** and the **MaxN**. The **R tolerance** is defined as the maximum $(R_{new} - R_{expected})/R_{expected}$. When the current $(R_{new} - R_{expected})/R_{expected}$ is smaller than the **R tolerance**, the new RS is assumed to be converged with an error in an acceptable margin. This parameter captures the maximum acceptable write error during weight updating. Another parameter **MaxN** is used to restrict the maximum number of 'prediction-write-verify' iterations to avoid the exceptional endless weight updating iterations. After obtaining the targeted RS, the current resistance is read and used as the initial state of the virtual device in the pulse parameter selection module. The module then returns the best parameter set among all provided options and applies a pulse to trigger the RS change. A read operation is taken to verify the new RS. The same procedure repeats until the stop conditions set by the **R tolerance** or the **MaxN** are met.

4.2.5 Customisation, usage scenarios and interface

With Python as the programming language, NeuroPack allows flexible, customised configurations at both algorithm and device levels and is engaged intimately with the community to encourage further evolution of the tool. NeuroPack can apply user-defined neuron models and learning rules at the algorithm level. At the algorithm level, NeuroPack is compatible with user-defined neuron models and learning rules: users can easily customise the neuron and the plasticity models by replacing the example core files with user-defined ones. At the device level, the flexible, empirical memristor model employed by NeuroPack can be reconfigured to model any type of memristor device with a simple set/change of parameters.

NeuroPack can assist users in two main scenarios. One is to serve as a supporting tool to investigate the sensitivity of the neuromorphic system performance to a particular parameter in neuromorphic computing tasks (Figure 4.2 **(A)**). In this scenario, users monitor how the behaviours of memristor devices change and explore how the classification accuracy and the loss are influenced, corresponding to different parameter values loaded from the configuration file. Another usage scenario is to test and verify system-level algorithms, mainly neuron and plasticity models (Figure 4.2 **(B)**). In this scenario, users configure the neural network topology by setting a user-defined core file or an example template that involves the targeted algorithm to be validated. Users can quickly visualise the results and check the changes in memristor device states and other internal variables for every single time step.

The embedded visualisation and analysis tool in NeuroPack contains a user-friendly GUI (can be seen in Appendix B, section 5) to display extracted key features and visualised data. Separated from the main panel of NeuroPack, this tool can work independently. When performing a classification task, NeuroPack stores internal variables for every single time step, including synaptic weights, membrane voltages, input spikes, output firing states, output errors, etc., in a Numpy (.npz) file. The analysis tool displays array-related variables such as synaptic weights in a colour map and neuron-related variables, including membrane voltages and output firing states, in line curves to show the time-dependent evolutions.

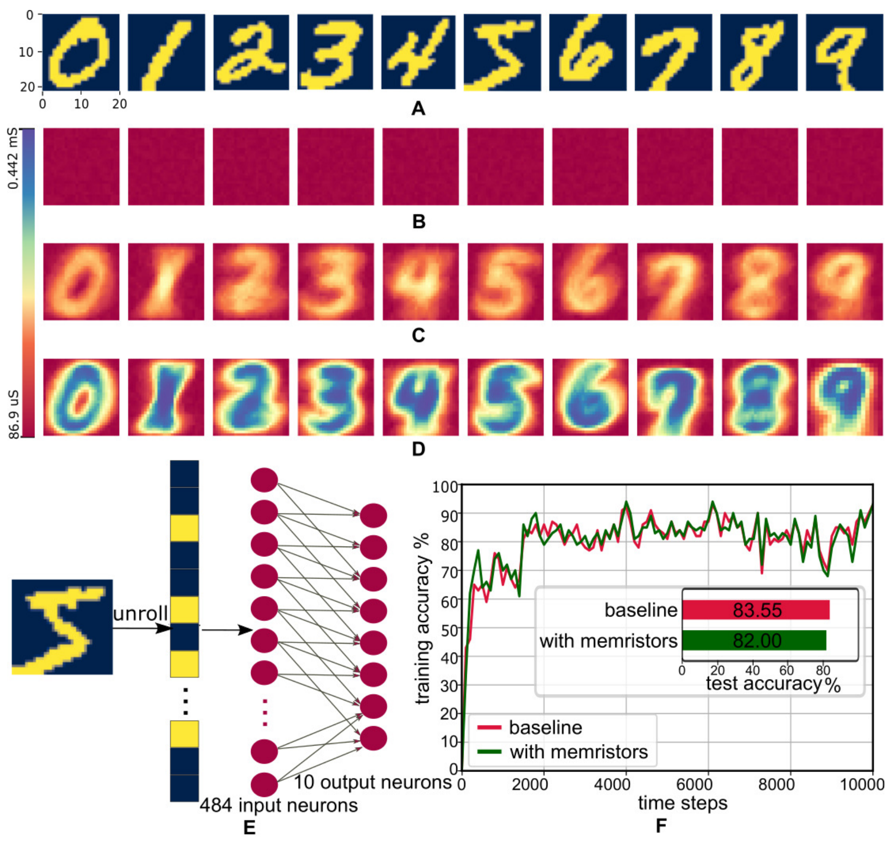


FIGURE 4.3: Simulation of a handwritten digit recognition task with a TiOx memristor-based neural network in NeuroPack. **(A)** Handwritten digits from the MNIST dataset cropped to 22 × 22 and binarised. Pixels 0 and 1 are coloured dark blue and yellow, respectively. **(B) - (D)** Memristor conductance sets before training **(B)**, after training for 2000 time steps **(C)** and after training for 10000 time steps **(D)** Memristor conductance spans from 86.9 uS to 0.442 mS, equivalent to the final memristor RSs after training ranging from around 2.26 kohms to 11.5 kohms.**(E)** Data processing procedure of the spiking neural network used to perform this handwritten digit recognition task. The neural network is a single-layer fully-connected SNN with a LIF neuron model. 22 × 22 -pixel input images are unrolled to 484-bit and represented as input spikes to be sent to 484 input neurons. This task incorporates a single-layer spiking neural network with 484 input and 10 output neurons. **(F)** Accuracy evolution curves during the training process with or without memristors. Curves are plotted every 100 samples for clarity. The memristor and the non-memristor version achieved the accuracy of 82.00% and 83.55%, respectively, in a separate 2000-sample test set. The image is from the author's publication [193].

4.3 Experiments

4.3.1 Demonstration: NeuroPack Simulation of a Handwritten Digit Recognition Task in the MNIST

This section showcases a handwritten digit classification task in the MNIST data set to validate NeuroPack.The original MNIST dataset images are cropped to 22×22 pixels and binarised (see example digits in Figure 4.3 **(A)**). The pixels used for background and digits are 0 and 1, respectively. Input neurons receive a 484-dimensional vector containing only 0 and 1 by unrolling a single image. For the input neurons, there is a spike when receiving 1 and no spike when receiving 0. To perform this classification task, a single-layer 484-input 10-output fully-connected winner-take-all spiking neural network with a leaky integrate-and-fire neuron model and gradient descent learning rule (see Figure 4.3 **(E)**) is employed. A total of 484×10 synapses are mapped to a 100×100 memristor array. 10000 samples were used to train the network. A balanced 2000 subset from the original MNIST test set was used to evaluate the training effect quickly (please see Appendix B, section 6 for more details). We use the abstract time step index rather than the actual time step as the x-axis to simulate and display the inference and the training. Upgrading the simulation with a volatility model is under planning due to the challenges in precisely controlling timing in software, though this functionality can better model the volatility characteristics of devices so that real-time dynamics are accounted for. Table 4.1 listed all NeuroPack parametric configurations used in the MNIST handwritten digit classification task. Memristor parameters used in this table are extracted using the method proposed by [202] on TiOx devices presented in [64], with the bias voltage from ±0.9V to ±1.2V and devices initialised around 11kΩ. Therefore, pulse options used for memristor RS updating are determined to be within those ranges. Estimated memristor operating ranges yielded by the model is between 2.23-12.8kΩ, with the bias voltage of ±1.2V and 12.5-18.9kΩ, with the bias voltage of ±0.9V. The resulting conductance is 5.3×10^{-5}-4.48×10^{-4}S, given the bias voltages ranging from ±0.9V-± 1.2V. The linear conversion from the memristor conductance to the synaptic weights can be expressed by the equation below to restrict the weights to be within [0, 1]

$$W = 2.53 \times 10^3 \times G - 0.1337$$

Memristors are initialised to have the resistance on 11kΩ with a variation no more than ±500 Ω before training. Memristor initial RSs correspond to small weights near the lower boundary of the device operating range, with the linear conversion of conductance into synaptic weights. Conductance sets before training, after training for 2000 time steps, and after training for 10000 time steps are displayed in Figure 4.3 **(B)-(D)**. During the training process, synapses connected between the digit pixels and labelled output neurons will increase the weights progressively, whereas the rest of the synaptic weights remain relatively small. As a result, targeted digits appear gradually in the conductance sets as the training continues. Linked by the linear mapping, the weight sets show the same tendency. Figure 4.3 **(F)** gives the accuracy evolution curves plotted for every 100 time steps. The network was fed with a separate test set with 2000 images after training, and 1640 out of 2000 gave correct inference results, counting a total classification accuracy of 82.00%. The baseline without using memristor models to store weights achieved a test accuracy of 83.55%. Therefore, the memristor model is not the accuracy bottleneck. Further investigation of improving the accuracy is given in Appendix B, section 7.

4.3.2 R tolerance Sensitivity Exploration

This subsection illustrates how to use NeuroPack to explore the relationships between the classification accuracy and the appropriately chosen values of the **R tolerance**, which is an issue that is intimately related to devices and programming protocols. Figure 4.4 **(A)** and **(B)** display the training accuracy revolution curves and the test accuracy results with different **R tolerance** values. The impact on the accuracy is trivial when the **R tolerance** value is small (within 1%). The training accuracy with respect to training sample numbers increases initially, reaches the peak, and then drops, with a larger **R tolerance** value. The corresponding test set accuracy also reflects this up-and-down tendency. The resistance changes of memristors driven by both stimulated and non-stimulated inputs with different **R tolerance** values are displayed in Figure 4.4 **(C)** to investigate what causes this accuracy evolution tendency. The baseline of virtual resistance values computed by the weight mapping scheme for a stimulated synapse (the one between input neuron 250 and output neuron 6 used as an example) as yielded by a non-memristor software synapse is given by the red line. The baseline displays a progressive decrease throughout the entire training with 10000 samples. In contrast, resistance updates are cut off increasingly early as the **R tolerance** increases. When the **R tolerance** is 0.1%, 1%, 2%, and 3%, the saturation occurs approximately after training $\sim$ 9000, 7000, 1000, and 100 time steps, respectively. The intuition is that if we fit the baseline trace with

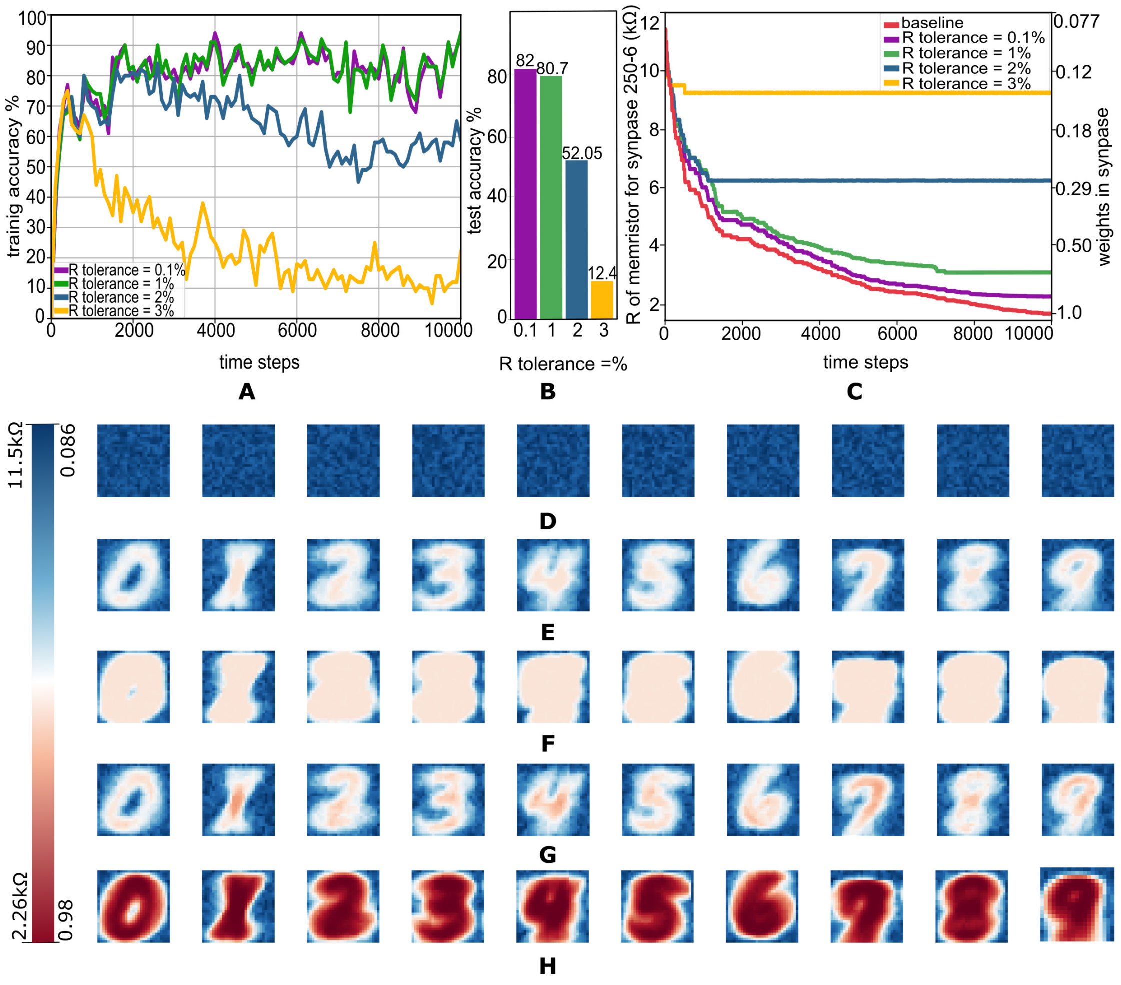


FIGURE 4.4: **R tolerance** sensitivity exploration. **(A)** and **(B)** Classification accuracy in the training and the test phases with different **R tolerance** values, respectively. **(C)** Resistance changes in an example memristor that represents a stimulated synapse (connected between the input neuron 250 and the output neuron 6) along the training process with different **R tolerance** values. Notably, virtual resistance values are used for the baseline. **(D)-(H)** Memristor RSs before training **(D)**, after 1000 time steps for the **R tolerance** of 2% **(E)** or 0.1%**(F)**, and after 10000 time steps for the **R tolerance** of 2%**(G)** or 0.1%**(H)**. Memristor RSs range from 2.26kΩ to 11.5kΩ, which corresponds to the weights from 0.98 to 0.086. The image is from the author's publication [193].

a smooth, continuous curve, its gradient continuously decreases because of the decreasing cost function gradient during the training process. That is to say, the required RS changes between two time steps reduce as training processes. Meanwhile, the **R tolerance** defined as $(R_{new} - R_{expected})/R_{expected}$ can be regarded as the cut-off ratio of memristor RS change. In other words, when the resistance change between two time steps is less than the **R tolerance**, the RS updates stop. Therefore, the memristor RSs stop updating earlier as the **R tolerance** value becomes larger. Figure 4.4 **(D) - (H)** displays the memristor RS sets before training **(D)**, after training 1000 time steps (with the **R tolerance** of 2% **(E)** and 0.1% **(F)**), and after 10000 time steps (with the **R tolerance** of 2% **(G)** and 0.1% **(H)**). The memristor RS sets before training **(D)**, after training 1000 time steps (with the **R tolerance** of 2% **(E)** and 0.1% **(F)**), and after 10000 time steps (with the **R tolerance** of 2% **(G)** and 0.1% **(H)**) are displayed by Figure 4.4 **(D) - (H)**. Three colour regions are clearly shown in the figures: blue for the high resistive range, white for the middle resistive range, and red for the low resistive range. Initial memristor RSs are in the high resistive range before training. Both versions with the **R tolerance** of 2% and 0.1% show distinguishable high and middle resistive regions after 1000 time steps, explaining the increasing tendency of the training accuracy before 1000 time steps in Figure 4.4 **(A)**. After 10000 time steps, the version with the **R tolerance** of 0.1% displays the distinguishable high, middle, and low resistive regions. In contrast, the low and the middle resistive regions are not distinguishable when the **R tolerance** is 2% because the resistance updates are cut off by a too-large **R tolerance** value. When the middle grows larger, the version with the **R tolerance** of 2% cannot accurately classify images because weights are all assigned with moderate values, regardless of the stimuli from different digits. As a result, the accuracy curves with large **R tolerance** values drop after certain points, as shown in Figure 4.4 **(A)**.

4.3.3 Bias Method Comparison for Selector-based arrays and Selectorless Arrays

Finally, a comparison between two biasing methods: a) that selected devices are applied with biasing voltages, which is commonly used in selector-based crossbar arrays), and b) that unselected devices in the same bitlines and wordlines are also applied with half-biasing voltages, as in selectorless crossbars, is presented. Figure 4.5 **(A)** shows the accuracy using these two different biasing methods. The training accuracy evolution curves show the same tendency with a $\sim$20% gap in between, which is further shown in the test accuracy bar chart. The resistance evolution curves of memristors representing a stimulated (connected between the input neuron 250 and the output neuron 6) and a non-stimulated synapse (connected between the input neuron 10 and the output neuron 6) are plotted to explore the cause of the performance gap. The baseline in red uses fully-software non-memristive synapses to store weights. The version using the biasing method of the selector-based arrays (in dark blue) exhibits the same decreasing trend in resistance changes as the baseline (in red). In

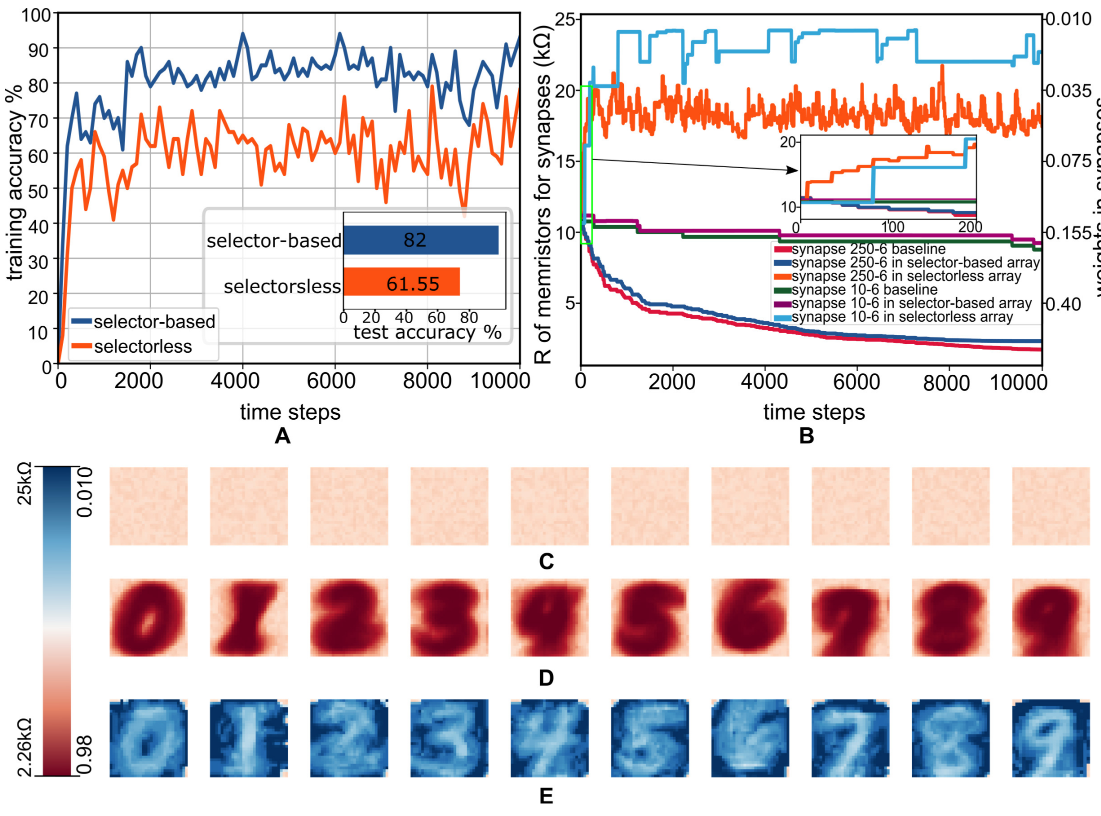


FIGURE 4.5: Classification with different biasing methods used in selector-based and selectorless memristor arrays. **(A)** Accuracy of different biasing methods for selector-based and selectorless arrays. **(B)** RS changes in memristors driven by mostly stimulative inputs (the one between the input neuron 250 and the output 6 as an example) and non-stimulative inputs (the one between the input neuron 10 and the output neuron 6 as an example). **(C) - (E)** give memristor RSs before training **(C)**, after training for biasing methods in selector-based **(D)** and selectorless **(E)** arrays. The colour bar indicates the memristor RSs ranging from 2.26k-25kΩ, which represents weights from 0.98 to 0.01 correspondingly. The image is from the author's publication [193].

contrast, the version with the biasing method of the selectorless arrays exhibits an increasing trend. Zooming into time steps 0 to 200, we notice unexpected resistance increases the selectorless scenario when there should be no resistance update. Unexpected resistance increases occur because devices connecting to silent neurons have been subjected to half-biasing voltages when the weights stored in the devices sharing the same wordlines/bitlines get updated. A single cycle of half-voltage bias does not cause a significant resistance increment. However, the effects of the single half-voltage biasing accumulate when many cycles are applied in the same time step as multiple devices share the same wordlines/bitlines. As a result, the accumulated errors eventually cause a large gap to the baseline. Non-stimulated synapses, both with the biasing method in the selector-based arrays (in purple) and without memristors (in green), stay around the initial values throughout the entire training process, whereas unexpected weight updates occur when using the half-voltage biasing method (shown in the trace coloured in light blue). However, in the given example of using the biasing method of a selectorless array, the resistance in memristors representing the stimulated synapse is still marginally lower than that of the non-stimulated synapse. Therefore, the NN with the biasing method used in a selectorless array can still learn the features of the input signals and do the classification correctly in some cases. Figure 4.5 **(C) - (E)** display the memristor RSs before and after training for both biasing methods. Due to the half-voltage biasing, the memristor operation range extends to 2.23k-28kΩ, corresponding to the new conductance ranging from 3.57×10^{-5}S to 4.48×10^{-4}S. The linear conversion of conductance into weights now has to be modified to be:

$$W = 2.42 \times 10^3 \times G - 0.0866$$

Memristor RSs are initialised as $\sim$11kΩ before the training starts. After training for 10000 time steps, the RSs of stimulated memristors with the biasing method of a selector-based array (Figure 4.5 **(D)**) drop to the low resistive range ($\sim$2.26kΩ), whilst the non-stimulated stay in the high resistive range ($\sim$11kΩ). The stimulated memristor RSs increase to a very high resistive range ($\sim$18kΩ) with the biasing method of a selectorless array; meanwhile, non-stimulated ones increase even higher to ($\sim$22kΩ). Thus, NeuroPack has assisted us in uncovering the perhaps surprising fact that the NN is still capable of distinguishing different handwritten digits in the MNIST substantially better than chance, even in the presence of unexpected invasive weights update, albeit with a very different weight distribution than that with the biasing method used in a selector-based network.

4.4 Discussion

This chapter presents NeuroPack, a versatile, Python-based fully-software algorithm-level simulator for memristor-empowered neuromorphic architectures. NeuroPack allows reconfiguring with user-defined settings at both the system and device levels. This framework can work as an independent tool to emulate neuromorphic architectures with different neuron, plasticity, and memristor models, different numbers or types of memristor devices, different neural network topologies, and different applications. Notably, Neuropack addressed the need for employing memristive technologies/models in developing SNN

TABLE 4.1: Parameters used in NeuroPack for MNIST handwritten digit classification task.

Global settings	
array size	100×100
array type	with selectors
read noise	0.1%
Neuron model	
threshold	25.16 or 24.16(for biasing method comparison only)
leakage	-0.3
Learning rule	
learning rate	3.5×10^{-6}
noise scale	10^{-6}
Memristor model	
A_p	0.21389
A_n	-0.81302
t_p	1.6591
t_n	1.5148
a_{0p}	37087
a_{0n}	43430
a_{1p}	-20193
a_{1n}	34333
Weights updating	
voltage	±0.9, ±1.1, ±1.2, ±1.2, ±1.2, ±1.2
pulsewidth	10^{-6}, 10^{-6}, 10^{-6}, 5×10^{-6}, 10^{-5}, 5×10^{-5}
R tolerance	0.1%
max update steps	5

topologies. The platform has been established and validated with a PC workstation (specs). Therefore, the absolute runtime depends on the hardware platform. However, the abstract runtime can be analysed. The simulation time scale is proportional to the number of synaptic connections rather than the neuron size. If the number of the synaptic connection increases by n orders of magnitude, the abstract runtime will also increase by n orders of magnitude, given there is unlimited memory to allocate. We further demonstrated an example of performing a handwritten digit classification task in the MNIST dataset with a single-layer SNN simulated in NeuroPack. We explored how the system classification accuracy is sensitive to the device- or programming protocol-related parameters such as the **R tolerance** in weight updating and the biasing methods employed in different array structures. We concluded: a) That the sufficient training efficiency to closely match the performance of an ideal model for this architecture and dataset is allowed even with a surprisingly max 1% tolerance in the RS, which indicates the potential competitive performance can be achieved by memristor-based systems without expensive precision circuits, at least in some scenarios. b) That it may be possible to retrieve meaningful information from the trained system even with unexpected weight updates due to the half-voltage biasing scheme employed in selectorless arrays. These explorations showcased how NeuroPack can assist users in designing and validating neuromorphic architectures and exploiting the potential of emerging nanoscale memory technologies in improving system performance. We envisage that users from the community can benefit from this tool to attain results that fit their needs quickly and efficiently by reconfiguring datasets, biasing methods, experimental parameters, device technologies and neuron connectivities.

Chapter 5

Text Classification in Memristor-based SNNs

5.1 Introduction

SNNs [188], referred to as the third generation of ANNs [205], have recently become an actively researched topic. Compared to traditional ANNs, SNNs feature a spike-based nature and temporal information coding scheme, which further bring the benefits of biological plausibility [206, 207], sparsity [208], and stochasticity [18]. All these features make SNNs more power-efficient than traditional ANNs. As a result, SNNs have been widely employed to perform different applications in different areas, including hand gesture detection [209, 210], gait detection [211], financial time series prediction [212], music composer classification [213], real-time signal processing [214], disease detection [215], and robotics [216, 217, 218]. Memristor further provides low-power solutions for SNNs implemention. Therefore, memristor-based SNNs have successfully demonstrated applications in pattern recognition and image classification [74, 75, 76, 83].

However, current usage of memristor-based SNNs still focuses on low-dimensional small-scale applications. For example, for text classification tasks, only a few works have performed text processing in memristor-based spiking neural networks: Design [162] used phase-change memory (PCM) to accelerate spiking neural network for language modelling, and Design [77] demonstrated a pattern generation task in recurrent spiking neural networks with PCM models using e-prop learning rule [86]. This is because expanding the usage of memristor-based SNNs to high-dimensional large-scale tasks is facing three major issues: high-dimensional inputs, costly training, and memristor no-ideality. To be more explicit, dealing with high-dimensional inputs requires allocating large memory, which is very costly in space efficiency. One solution would be using dimensional reduction technology. For example, word embeddings (e.g. GloVe [84] and Word2vec [85]), are commonly employed dimensional reduction models to represent words than one-hot vectors in text classification tasks to improve space efficiency and extract semantic relationships between words. A word embedding layer is used as a lookup table in traditional ANNs to produce a dense representation of the input texts. Pre-trained word embeddings are typically fine-tuned later in training along with other network parameters to achieve better performance. They are treated as linear layers utilising back-propagation (BP) [107] in training. However, the dense word representation is supposed to be transformed into spike trains as the network's inputs in SNNs [219, 220]. So far, previous publications provide no theoretical foundation for training word embeddings in spiking neural networks. This brings the first issue of expanding the usage of memristor-based SNNs: how to efficiently employ dimensional reduction technologies in memristor-based SNNs? Secondly, The expensive training of memristor-based spiking neural networks is another major obstacle. To begin, gradient-based learning procedures, regardless of local (e-prop) or non-local (e.g. surrogate gradient descent [198, 221]) learning rules are used, necessitating averaging of the accumulated errors over the spike train window used to represent a single numerical value. This requires considering the impact of every single spike on updating weights. It is inefficient in processing speed and space efficiency, particularly when the spike trains used to represent a single numerical value are lengthy and memristors are included in the design. Finally, basic read and write operations on memristor arrays can also introduce variances due to read noise and write variation. One possible solution to avoid these obstacles is to execute the training in a software simulator using accurate memristor models. This solution provides a faster data process and intuition for system performance than direct training in hardware. Reference [193] developed an algorithm-level simulator for memristor-based spiking neural networks in memristor models, and Design [83] also utilised this solution to simulate recurrent spiking neural networks in PCM models.

This chapter tries to expand the usage of memristor-based SNNs to high-dimensional large-scale applications by solving the issues of high-dimensional inputs, costly training, and memristor non-ideality. To do so, NeuroPack in Chapter 5 has been extended with Pytorch [91] to a bespoke simulation framework with GPU acceleration support. We first take two paths to train memristor-based spiking neural networks to leverage this framework in text classification tasks. One is converting a pre-trained ANN to its equivalent SNN, and another is directly training an SNN using an ANN-based learning rule. Both paths train the network using gradient descent-based learning rules, only taking the spike rate into account to improve the computational speed and the space efficiency. In this method, we can also add a dimension reduction model and train a word embedding layer as a linear layer as it is in a standard ANN. Then, we demonstrate the first sentiment analysis task in the IMDB movie reviews dataset [92] to validate both approaches. From what we are aware of, this is the first demonstration of a text classification task completed in a spiking neural network using a realistic memristor model. Finally, we summarise systems classification accuracy taking both paths and explore how global factors impact system performance.

The contributions of this work are summarised below:

1. Developing a simulation framework with an empirical memristor model to demonstrate the first text classification task in spiking neural networks with a realistic memristor model.
2. Presenting and benchmarking two approaches to obtain trained memristor-based spiking neural networks for text classification tasks.
3. Investigating system sensitivity to global parameters.

To achieve these goals, in this chapter, we first introduce the overview of the two paths we take to obtain a trained memristor-based SNN in subsection 5.2.1 with details shown in subsection 5.2.2 and 5.2.3. The method to add the memristor model to the simulation framework is introduced in subsection 5.2.4 and the weight updating scheme in subsection 5.2.5. Section 5.3 displays all experimental results for the text classification task in the simulation framework with memristor models, and section 5.4 summarises the paper.

This work has led to the publication of a pre-printed paper [222].

The source code of the work presented in this chapter can be found in the Github repository:
https://github.com/hjq310/text-classification-in-memristorsnn

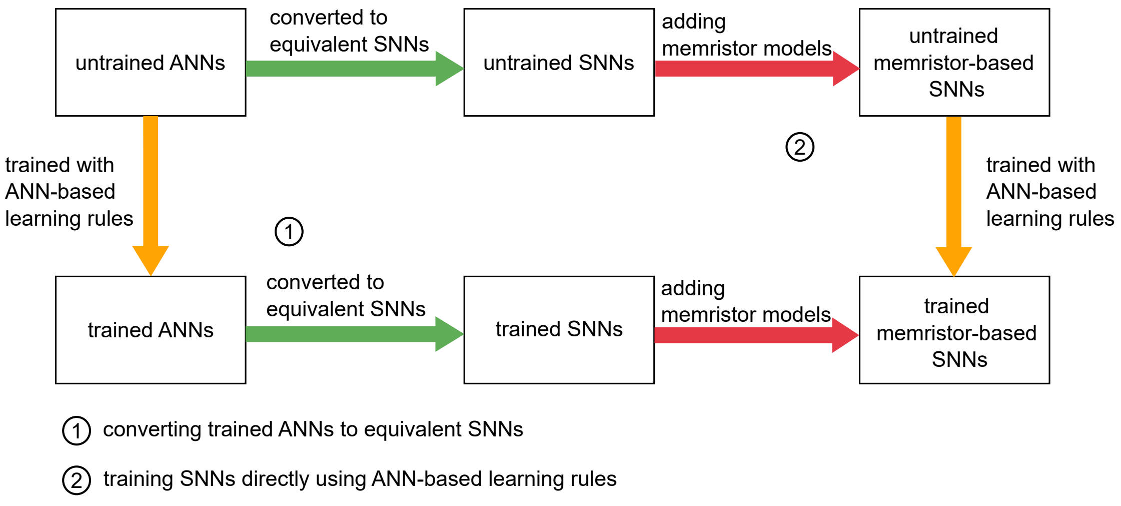

untrained ANNs
converted to equivalent SNNs
untrained SNNs
adding memristor models
untrained memristor-based SNNs
trained with ANN-based learning rules
②
①
trained with ANN-based learning rules
trained ANNs
converted to equivalent SNNs
trained SNNs
adding memristor models
trained memristor-based SNNs
① converting trained ANNs to equivalent SNNs
② training SNNs directly using ANN-based learning rules

FIGURE 5.1: Two approaches to obtaining trained SNNs: approach 1 starts by training ANNs (coloured orange), transforming the trained ANNs to equivalent SNNs (coloured green), and then mapping SNN weights to memristors (coloured red); approach 2 firstly converts untrained ANNs to their equivalents (coloured green), adds a memristor model (coloured red), and then trains memristor-based SNNs directly using ANN-based learning rules (coloured orange). The image is from the author's publication [222].

5.2 Methods

5.2.1 Methodology Overview

The fundamental difference between ANNs and SNNs is the way of conveying information: ANNs use continuous values, whereas SNNs use 0-or-1 spikes. Therefore, it is natural to try to find the relations to map continuous values to spikes to bridge the gap between two neural network architectures. Reference [223] revealed that the spike rates of an SNN are proportional to the ReLU [224] activation outputs of the equivalent ANN with an error term that can be ignored in shallow networks. The ReLU activation outputs in the equivalent ANN are always non-negative in a memristor because inputs and weights are restricted within [0, 1] in a memristor-based SNN. As a result, an SNN's spike rates are proportional to the input currents. Reference [225] further proved that the larger the membrane voltage threshold, the smaller the error term when

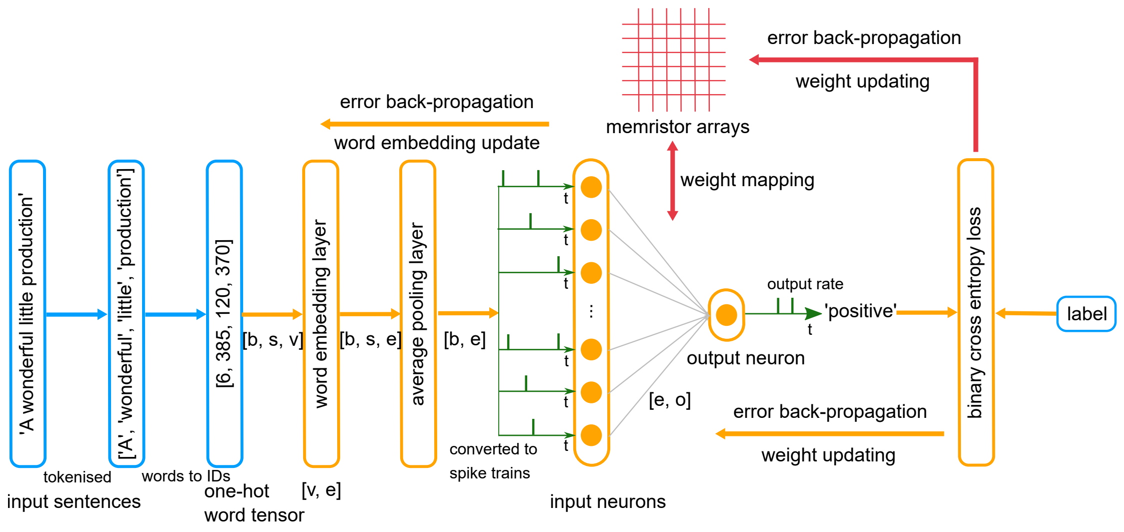


FIGURE 5.2: Simulation framework workflow for performing the sentiment analysis task in the IMDB review dataset. Inputs, ANN structure, SNN-specified structure, and memristor-related modules are coloured blue, orange, green and red, respectively. b, s, e, and o represent batch size, sentence length, word embedding dimensions, and output dimensions, respectively. To demonstrate input pre-processing steps, we use a movie review, 'A wonderful little production,' as an example: firstly, tokenising input sentences to lists of words ('['A', 'wonderful', 'little', 'production']' in the example); secondly, converting the word lists to lists of word IDs ('[6, 385, 120, 370]' in the example); finally, padding the sentences in the same batch to have the same length, transforming the lists of words to a one-hot vector, and packing sentence vectors into an input representation tensor with the size of ($b \times s \times v$) to represent the input sentence. Notably, transforming word ID lists to one-hot representation is skipped if Pytorch is used. The image is from the author's publication [222].

'reset by subtraction' [1] is applied for spiking neurons. In addition, weight normalisation [226] and adaptive threshold [227] are also widely utilised in ANN-to-SNN conversion to match the accuracy. These previous works form the theoretical basis for the two paths to obtaining trained memristor-based SNNs proposed taken by this work.

Figure 5.1 shows the two paths taken by this work to obtain a trained memristor-based SNN. With an untrained ANN, approach 1 firstly trains the ANN using standard ANN-based learning rules, then transforms the trained ANN to an equivalent SNN; approach 2 starts from the conversion of an untrained ANN into an SNN before training with (appropriately modified/selected) ANN-based learning rules. Memristor models are added to represent synapses after the

[1]'reset by subtraction': subtracting a fixed number from the membrane voltage when resetting it after firing.

trained SNN is ready. This step is as simple as mapping the weights in memristor arrays as conductance in approach 1, whereas approach 2 requires memristor resistance states (RSs) to be trained alongside the entire SNN.

Now we walk through the simulation framework workflow for performing the sentiment analysis task to explain how the two paths are implemented. Figure 5.2 displays the diagram-level structure of the network we used in the sentiment analysis task in the IMDB reviews dataset. After being tokenised using the basic English tokeniser and converted to word IDs, sentence one-hot vectors are padded to have the same length and packed into an input one-hot representation tensor with the size of ($b \times s \times v$) (where b, s, v are batch size, sentence length, and vocabulary size, respectively) is used to generate a dense word representation with the size of ($b \times s \times e$) (where e is the word embedding dimension) by being fed to a word embedding layer, followed by an average pooling layer that squeezes the sentence dimension to obtain an averaged sentence representation across the entire sentence with the size of ($b \times e$).

In approach 1, the averaged sentence representation with the size of ($b \times e$) is fed straight to a linear layer whose input and output neuron numbers are e and o, respectively. Specifically, only one output neuron is used in the sentiment analysis task because this is a binary classification task with two categories: 'positive' or 'negative'. Once the inference results are obtained, the binary cross-entropy loss is computed, and the errors are transmitted backwards to update weights in the linear and word embedding layers. The conversion to an SNN starts after the training of the ANN completes. Firstly, Poisson spike trains are generated by using the averaged sentence representation and delivered to the single-layer spiking neural network. The leaky integrate-and-fire neuron model [89] is used in this task. Next, the weights in the SNN are mapped as the conductance of memristor devices and loaded from the memristor arrays when inferences are executed. The output neuron membrane voltages are computed using the neuron model's mathematical expression, and the firing states are updated by comparing the voltage to the threshold. The final inference results are given by the spike rates within a time window: 'positive' if the spike rate is higher than 50%; 'negative' otherwise.

Approach 2 also transforms the averaged sentence representation to Poisson spike trains, and memristor arrays are randomly initialised before the training starts. During the training, the errors are back-propagated with the same ANN-based learning rule used in approach 1, and the expected weight changes are converted to the memristive RSs to be updated. The errors are further back-propagated to the word embedding layer to generate new spike representations for the following inferences.

In the next two subsections, more details on the implementations of the two approaches in the simulation framework will be revealed.

5.2.2 Approach 1: Converting a Pre-trained ANN to an SNN

When training the ANN using approach 1, weights in the linear and word embedding layers must be restricted in the range [0, 1]. With this constraint, weights can be expressed using the following equation when converting the ANN to the SNN:

$$w = \frac{G - G_{min}}{G_{max} - G_{min}} \tag{5.1}$$

Where G, G_{min}, G_{max} represent the current, the minimum, and the maximum memristor conductance, respectively. The averaged sentence representation is also constrained to be within [0, 1]. Therefore, the continuous values in the averaged sentence representation can be used as firing rates to generate spike trains when the ANN is transformed into the SNN. Reference [219] presented a strategy using the following equation to determine whether a spike should be released when generating a Poisson spike train:

$$P_t = \begin{cases} 1 & \text{if } x_c > x_{random,t} \\ 0 & \text{otherwise} \end{cases}, \quad 0 \leq t \leq T \tag{5.2}$$

Where P_t denotes the spiking state determined by the averaged sentence representation in the given time step t in the spike train with T time steps to encode a single continuous value, x_c is the continuous value being encoded in the averaged sentence representation, and $x_{random,t}$ is the random value generated in the time step t within [0, 1]. Before the binary word embedding conversion, an initialised random-value matrix with the size of $(e \times T)$ is created. Poisson spike trains stored in a matrix with the size of $(e \times T)$ used as the inputs to the single-layer SNN are generated by comparing the averaged sentence representation to the random-value matrix.

The outputs of the linear layer derived by the equation $y = \sum W_s x_c$ are always positive because the weights in the linear layer W_s and the averaged sentence representation x_c are both non-negative while training the ANN. When using the sigmoid function to convert the continuous-valued outputs to probabilities, a negative constant offset C needs to be introduced to the ANN outputs. Without the negative offset C, the input of the sigmoid function is never negative. Therefore, the output of the sigmoid function will always give a probability not less than 50%. To make the probability transform functional, the term passed to the sigmoid function is now $(y + C)$ rather than y. The inference result lies in the decision boundary that separates two categories when $(y + C) = 0$. By solving this equation, $C = -\sum \bar{W}_s \bar{x}_c = -0.5 \times 0.5 \times e$ where $\bar{W}_s$ and $\bar{x}_c$ are both the theoretical averaged values (-0.5 and -0.5, respectively). However, when converted to the SNN, the criterion for dividing categories is the output spike rates. Therefore, an offset is not required to be added to the SNN because the output neuron spike rates are already probabilities and can be used directly as the criterion for dividing categories without being connected to a sigmoid function.

The membrane voltage threshold is an SNN-exclusive parameter that needs to be chosen when converting an ANN to an SNN to match the classification accuracy. Information is lost when membrane voltages are much higher or lower than the threshold [227]. Therefore, choosing a threshold value when the membrane voltage is at half its theoretical maximum value is a good start to avoid catastrophic information loss. The membrane voltage range in converted SNNs is known since both the weights and the word embeddings are restricted to the range [0, 1]. Without considering the clipping caused by the threshold, the output neuron has the maximum theoretical membrane voltage when all inputs and weights are set to "1" along the entire spike train. Therefore, we can express the estimated threshold as $0.5 \times e$. Using this value as the threshold ensures that the maximised inputs can result in the maximum firing rate and that the output neuron accumulates membrane potential in a way proportional to the intensity of the inputs without being saturated or clipped under any initial conditions. Better performance could be achieved by fine-tuning the threshold from the estimated value.

5.2.3 Approach 2: Training an SNN directly

Due to the non-differentiable nature of membrane voltages, training SNNs using ANN-based gradient descent learning algorithms [87] can be challenging. Employing surrogate derivatives [198, 221] with straight-through estimators [199, 200, 201] is a common solution to solve this issue. This work presented an alternate method derived from the ANN-based gradient descent mechanism without taking the derivative of the non-differentiable function. A leaky integrate-and-fire (LIF) neuron model with reset by subtraction is employed in this work using the equation below:

$$\begin{aligned} V_t &= V_{t-1} + \sum W_s x_t - Z_{t-1} V_{th} \\ Z_t &= h(V_t - V_{th}) \end{aligned} \tag{5.3}$$

Where V_t, x_t, and Z_t correspond to membrane voltages, input spikes and output spikes at time step t, respectively. V_{th} denotes the membrane voltage threshold, W_s is the weights in the single-layer SNN, and $h(x)$ is the Heaviside step function. We employ the binary cross-entropy cost function with a sigmoid layer for this binary classification task:

$$E = -(\hat{y} \log y + (1 - \hat{y})(1 - \log y)) \tag{5.4}$$

Where E denotes the binary cross-entropy loss, $\hat{y}$ and y represent output labels and the probabilities predicted by the network that the current sample belongs to the labelled category, respectively.

In this work, the Adaptive Gradient Algorithm (Adagrad) [228], a gradient-based algorithm with an adaptive learning rate dependent on the past gradient, is chosen as the learning rule. The mathematical expression of Adagrad is shown as follows:

$$\begin{aligned} s &\leftarrow s + g^2 \\ \theta &\leftarrow \theta - \eta \frac{g}{\sqrt{s} + \epsilon} \end{aligned} \tag{5.5}$$

Where g is the gradient of the cost function over the parameter θ, s is a state variable equals to the sum of the square of the past gradients, and ϵ is a small value to avoid dividing by 0.

Now we walk through the process to obtain the final expression of the weight updates. The output firing rate is always non-negative, as explained in subsection 5.2.2. Therefore, an offset equal to -0.5 must be added to the output firing rate before being delivered to a sigmoid function. Passing the output firing rate with a negative offset to a sigmoid function yields the output probabilities of two categories:

$$a = r + C \tag{5.6}$$

$$y = \frac{1}{1 + e^{-a}} \tag{5.7}$$

Where r is the output neuron firing rate, C is the negative offset, and a is a temporary variable used to transform the firing state.

As proven in [225], the output firing rates of an SNN are proportional to the equivalent ANN activation outputs in the equivalent ANN. We further proved that the output firing rates in a memristor-based SNN are proportional to the corresponding ANN outputs as expressed in the equation below (please see the derivation in Appendix C, section 1), provided that the outputs are always non-negative:

$$r = \frac{V_c}{V_{th}} - \frac{V_t - V_0}{V_{th} \cdot T} \tag{5.8}$$

Suppose the equivalent ANN is expressed with the equation shown below:

$$V_c = \sum W_s x_c \tag{5.9}$$

Where V_0 is the initial membrane voltage of a spiking neuron when a new inference starts, and V_c and x_c are continuous-valued ANN outputs and inputs, respectively. V_0 equals 0 in this work because the membrane voltage accumulator is reset before processing a new sample. In Equation 5.8, the error term $V_t/(V_{th} \cdot T)$ depends on the final membrane voltage V_t at the end of the spike window T, and V_c is the continuous value that the spike train within the spike window T represents. Intuitively speaking, V_t implicitly depends on V_c. Therefore, the error term $V_t/(V_{th} \cdot T)$ is somehow weakly related to V_c, and this weak relation can potentially introduce an error in the ANN-to-SNN conversion. This error can be accumulated in a deep neural network, but it can be ignored in a shallow network [225]. By ignoring the weak dependency of the error term to V_c, we can rewrite the function of the output firing rate if we use two constants α and β to replace the error term and the constant coefficient:

$$r = \alpha V_c + \beta \tag{5.10}$$

The derivative of r_t over V_c is $dr/dV_c = \alpha$. The gradient of the cost function over the weights in the single-layer SNN can be computed by using the chain rule:

$$\begin{aligned} \frac{dE}{dW_s} &= \frac{dE}{da} \cdot \frac{da}{dV_c} \cdot \frac{dV_c}{dW_s} \\ &= (y - \hat{y}) \cdot \alpha \cdot x_c \end{aligned} \tag{5.11}$$

Likewise, by further propagating the errors backwards, the word embedding matrix can be updated:

$$\begin{aligned}\frac{dE}{dW_e} &= \frac{dE}{da} \cdot \frac{da}{dV_c} \cdot \frac{dV_c}{dx_c} \cdot \frac{dx_c}{dW_e} \\ &= (y - \hat{y}) \cdot \alpha \cdot W_s x_e \end{aligned} \tag{5.12}$$

Where W_s gives the parameters in the word embedding matrix, and x_e is the one-hot representation of input words.

The weight change is proportional to $g/(\sqrt{s} + \epsilon)$ in Adagrad. g and $\sqrt{s}$ share the same scaling factor α. Therefore, the final expression used to update weights can be simplified by cancelling out the α:

$$g_s = (y - \hat{y}) \cdot x_c \tag{5.13}$$

$$g_e = (y - \hat{y}) \cdot W_s x_e \tag{5.14}$$

Our solution skips the step of accumulating errors across the spike window and increases the computing speed and space efficiency compared to collecting errors across the entire spike train via surrogate gradient descent, which is represented by the equation below:

$$\begin{aligned}\frac{dE}{dW_s} &= \frac{dE}{da} \cdot \frac{da}{dZ_t} \cdot \frac{dZ_t}{dy_t} \cdot \frac{dV_t}{dW_s} \\ &= (y - \hat{y}) \cdot \frac{1}{T} \sum^{T} h'(V_t - V_{th}) x_t \end{aligned} \tag{5.15}$$

The gradient computed with surrogate derivative (Equation 5.15) can be simplified to $(y - \hat{y}) \cdot x_c / (2V_{th})$, if a binary activation [201] shown in Equation 5.16 is employed to replace the non-differentiable Heaviside step function and a threshold equal or larger than the half of the maximum membrane accumulator capacity is ensured to avoid information loss. The simplified gradient expression is shown in Equation 5.13 and 5.14 by cancelling out the constant coefficient $1/(2V_{th})$.

$$h'(V_t - V_{th}) = \begin{cases} \frac{1}{2V_{th}} & \text{if } 0 < V_t < 2V_{th} \\ 0 & otherwise \end{cases} \tag{5.16}$$

5.2.4 Adding Memristor Models

The statistical memristor model presented by [88] is used in this study to predict how memristors behave during the training or inference phases of memristor-based SNNs. The memristive switching rates are expressed by the equation reproduced from [88] below for convenience:

$$\frac{dR}{dt} = \begin{cases} A_p(-1+\exp(|v|/t_p))h(r_p(v)-R)(r_p(v)-R)^2 & \text{if } v > 0 \\ A_n(-1+\exp(|v|/t_p))h(R-r_n(v))(R-r_n(v))^2 & \text{otherwise} \end{cases} \tag{5.17}$$

Where $A_{p,n}$ is the scaling factor, the expression $(-1+\exp(|v|/t_{p,n}))$ illustrates the exponential growth between the switching rate and the bias voltage v. The rest of the equation illustrates the dependency on the current memristive RSs and the dynamic upper/lower boundaries $r_{p,n}$ expressed as the equation below with fitting parameters $a_{0p,n}$ and $a_{1p,n}$:

$$r_{p,n} = a_{0p,n} + a_{1p,n}v \tag{5.18}$$

All parameters can be extracted using characterisation instrumentations [80, 169] with the method presented by [202] in practice. Knowing the current memristive RSs and the bias voltages, this framework uses these parameters to predict the memristor RSs. Implemented in Pytorch, all memristor RSs are pulsed and read simultaneously. Provided the pulse voltage $\mathbf{v}$ and the pulse-width $\mathbf{pw}$, the duration is transformed to iteration times using the equation $\mathbf{N} = \mathbf{int}(\mathbf{pw}/dt)$ to update and memristive RSs with the changes $\Delta R = \sum^{N}(dR/dt)$ in loops.

5.2.5 Weight Updating Scheme

A critical step is to convert the expected weights to the expected memristive RSs after acquiring the expected weights. Equation 5.2 can be rewritten to the expression of the expected memristive RSs based on the expected weights:

$$R = \frac{1}{w(\frac{1}{R_{min}} - \frac{1}{R_{max}}) + \frac{1}{R_{max}}} \tag{5.19}$$

Where R, R_{min} and R_{max} are the expected memristive RS, the lower and the upper RS boundaries, respectively. Bias voltages and the current memristive

RSs govern the non-linear memristive switching dynamics, as shown in Equation 5.17. As a result, it is challenging for pulsing memristors to be in specific RSs. Chapter 4 suggested a ”predict-write-verify” cycle to efficiently find the parameter set (typically including bias voltages and pulse-width) that can produce memristive RSs close to the expected [193]. With multiple sets of pulsing parameters provided, all resulting memristor RSs are firstly predicted, and the one that contributes to the shortest distance to the expected RSs is selected. Next, memristors are pulsed with the selected parameter set, and the actual new RSs are verified. Repeat this process until the distance between the expected and actual RSs falls within the acceptable margin specified by the parameter **R tolerance** defined as $(|R_{expected} - R_{actual}|)/R_{expected}$. Another parameter **maxN** restricting the maximum iteration times is also used as a stop condition to avoid possible infinite loops.

When applying this weight updating scheme, selecting pulsing parameter sets is a critical design decision. Assume only the pulsing parameter sets that can lead to small RSs changes are provided. The steps may be too small for the memristors to reach the expected values before meeting the stop condition set by the **maxN** if the distances between the actual and the expected RSs are large. In another extreme condition that only parameter sets that allow significant RS changes are available, the targeted RSs may never be reached. Matching the estimated RS changes that the selected parameter sets can result in with the required RS changes is, therefore, a viable solution. The needed RS changes for approach 1 are often rather large. Therefore, parameter sets that can result in significant and tiny RS changes are both necessary: the former makes RSs converge to the expected states quickly, whereas the latter finely tunes the RSs. Parameter sets that lead to relatively tiny RS changes are only needed in approach 2 because the required RS changes in each training step are relatively small. For both approaches, parameter sets that can result in RS updates smaller than the **R tolerance** range are required to ensure that the weight updating can fulfil the stop condition set by the **R tolerance**.

5.3 Experiments

UNIVERSITY OF SOUTHAMPTON Faculty of Engineering and Physical Science Optoelectronics Research Center Memristor-based Spiking Neural Networks by Jinqi Huang ORCiD: 0000-0003-3913-0067 A thesis for the degree of Doctor of Philosophy October 2022

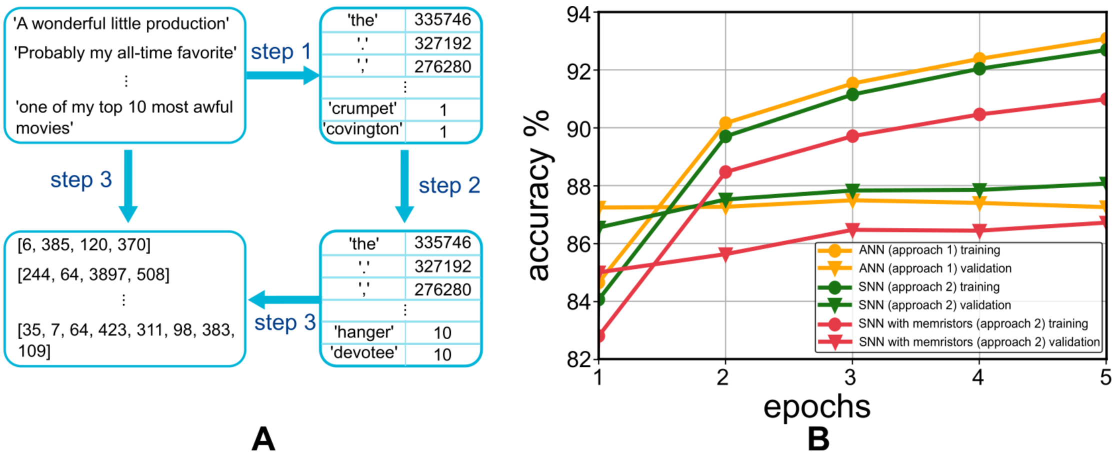


FIGURE 5.3: **(A)** Creating a vocabulary using the training samples. Step 1: Build a look-up table to store word frequency. Step 2: Delete the words whose frequency is smaller than 10 to reduce the vocabulary size. Step 3: Transform all samples to lists of word IDs with the aid of the vocabulary. If the word cannot be found in the vocabulary, tag it as ′[unk]′. Lists of word IDs are padded with the ′[pad]′ token to ensure the same length when packed into the same batches. **(B)** Training and validation accuracy evolution curves in the ANN (approach 1), the SNN (approach 2), and the SNN with memristor models (approach 2). The image is from the author′s publication [222].

5.3.1 Experiment Overview

In this section, we showcase a sentiment analysis task with the IMDB movie reviews dataset by taking the presented two approaches to obtain a trained memristor-based SNN. There are 25k highly polar movie reviews training samples and 25k test samples in the dataset. Inspired by a GitHub project [229], training examples are used to construct a vocabulary to pre-process the input samples (Figure 5.3 **(A)**). The word frequency in the training samples is counted, and those words that occur over 10x times are put into the vocabulary. This procedure constructs a vocabulary with 20473 words. Next, the original training set is split into a sub-training set with 17.5k samples and a validation set with 7.5k samples. An input review is then tokenised with the basic English tokeniser and transformed into a word ID vector. The neural network is then fed with the one-hot representation of the word ID vector to start an inference. The word embedding layer loads pre-trained GloVe word embeddings [84] with 100 dimensions. Please refer to Section ′Methods′ for the training and inference procedures. The network is trained for 5 epochs in this experiment. After training with 17.5k samples from the training set for an epoch, the training is turned off, and 7.5k samples from the validation set are fed into the network to validate the training effect for this epoch. Before starting another epoch, samples in the training and validation set are shuffled. Notably, shuffling only breaks the dependency between training samples but not introduces variations in testing with different neural network architectures, because (1) the test set was not shuffled; (2) the random seed for shuffling is fixed. After training, the trainable network parameters that result in the smallest validation loss are reloaded to the network for testing. The network then receives 25k samples from an independent test set. Table 5.2 summarises the experiment settings for the two approaches, Figure 5.3 **(B)** shows the accuracy evolutions during the training procedure, and Table 5.1 shows the final test accuracy. The ANN in approach 1 provides a baseline test accuracy of 86.02%. According to [229], more complicated models, such as transformers [158], need to be employed to improve the classification accuracy. Table 5.1 shows that in Approach 1, the test accuracy degrades from an ANN to an SNN by 0.13% and from SNN to memristor-based SNN by 0.01%. In method 2, the test accuracy deterioration from ANN to SNN is 0.1 %, whereas adding memristor models further reduces the test accuracy by 1.06%.

TABLE 5.1: Test accuracy for approach 1 and 2.

	ANN	SNN	SNN with memristors	SNN	SNN with memristors
approach	1	1	1	2	2
accuracy (%)	86.02	85.89	85.88	85.92	84.86

5.3.2 Results Analysis of Approach 1

This subsection analyses the results from approach 1. Figure 5.4 **(A)** displays the classification accuracy dependency on the length of a spike train to represent a single numerical value. An 'increased-and-saturated' trend can be observed along with the increase of the spike train length because of the stochasticity of spike trains. Theoretically speaking, longer spike trains can represent the numerical values more accurately. However, increasing the length stops improving the spike representation accuracy when the length is long enough. Therefore, the classification accuracy saturates. Longer spike trains also sacrifice more runtime to process over the entire spike train for a single numerical value. T=1k is chosen for the standard configuration to balance the system performance and the computation efficiency.

Figure 5.4 **(B)** depicts the system sensitivity to the **R tolerance** when approach 1 is taken. In theory, larger **R tolerance** values increase the maximum possible errors between the expected memristor RSs and the actual values when exporting trained weights as memristor RSs, causing potential performance degradations. However, the system taking approach 1 is robust to large **R tolerance**

TABLE 5.2: Parameters used in the sentiment analysis tasks baseline configuration for Approach 1 and 2. Different parameter values for different approaches are highlighted in light grey.

Parameters	Approach 1	Approach 2
Training set size	17.5k	17.5k
Validation set size	7.5k	7.5k
Test set size	25k	25k
Vocabulary size	20473	20473
Embedding dimension	100	100
Output dimension	1	1
Batch size	1	1
Offset	-25	-0.5
T	1k	1k
V_{th}	50	56.75
η	0.05	0.05
ϵ	1×10^{-8}	1×10^{-8}
Array size	10×10	10×10
A_p	0.21389	0.21389
A_n	-0.81302	-0.81302
a_{0p}	37087	37087
a_{0n}	43430	43430
a_{1p}	-20193	-20193
a_{1n}	34333	34333
t_p	1.6591	1.6591
t_n	1.5148	1.5148
Positive pulse magnitude (V)	0.9	0.9
Positive pulse duration (us)	1, 2, 10, 20, 50, 100	1, 2, 10, 20, 50
Negative pulse magnitude (V)	-1.2	-1.2
Negative pulse duration (us)	1, 2, 10, 20, 100, 1000, 2000, 5000	1, 2, 10, 20, 100
dt (ms)	1	1
R tolerance	0.05%	0.05%
MaxN	5	5

values, as shown in the bar chart. To investigate the reasons, the weight error standard deviation with different **R tolerance** values is plotted as shown in Figure 5.4 **(C)**. The weight error standard deviation is no more than 0.06 even with the **R tolerance** of 20%. This shows that the system can accept variations when exporting weights into memristor arrays in converting a trained ANN to a memristor-based SNN.

Similarly, approach 1 also shows tolerance to the read noise introduced by instrumentation during the read operation for memristor arrays (Figure 5.4 **(E)**). This framework simulates the read noise by adding a random value within a certain scale to the memristor RSs. For example, the read noise is 10% means that the RS read errors are within $\pm 10\%$ of the actual memristor RSs (Figure 5.4

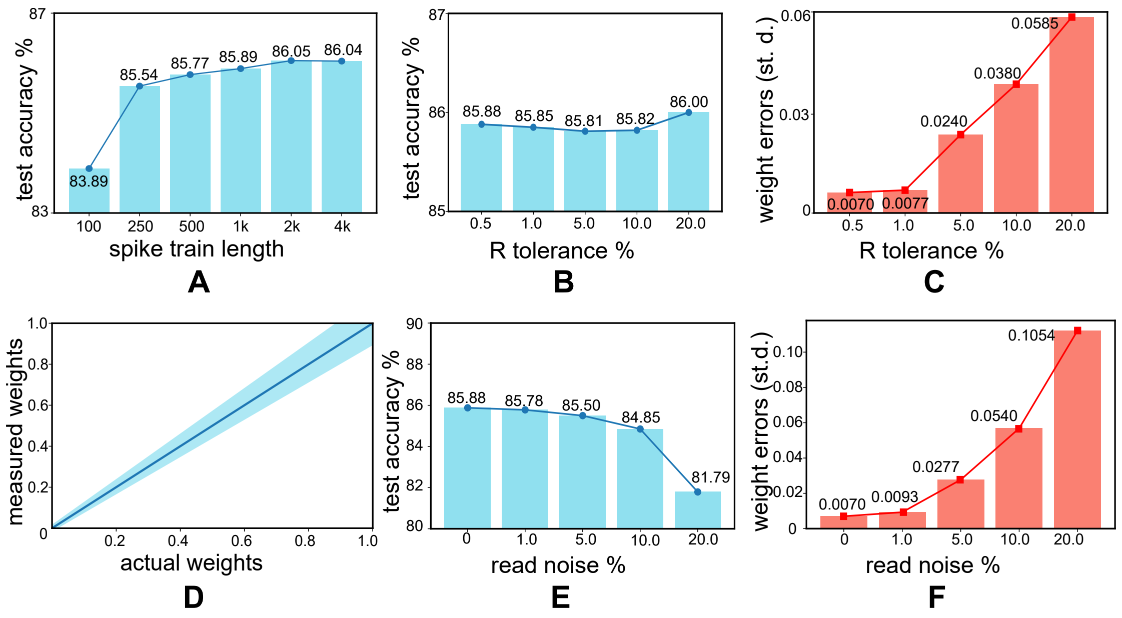


FIGURE 5.4: Result analysis for converting an ANN into a memristor-based SNN.**(A)** Test accuracy vs spike train length, where the spike train represents a single numerical value. **(B)** Test accuracy vs the **R tolerance** values when exporting weights as memristor RSs. **(C)** Weight standard deviations along with the increase of the **R tolerance** when exporting weights as memristor RSs. **(D)** The measured weights along with the actual weights with 10% read noise. **(E)** The test accuracy dependency on the read noise. **(F)** Weight standard deviations along with the increase of the read noise. The image is from the author's publication [222].

(D)). Both the prediction stage of the weight updating scheme and the inference phase are affected by the read noise. Even with 20% read noise, which implies the discrepancy between the measured and the actual memristor RSs is up to 20% in the worst scenario, the system can still achieve 81.79% classification accuracy. Figure 5.4 **(F)** shows the weight errors standard deviation along with the increase of the read noise. The weight error standard deviation introduced at 20% read noise is around 0.105 (Figure 5.4 **(F)**).

5.3.3 Results Analysis of Approach 2

Now we investigate how the weight updating process is affected by the **R tolerance** when utilising approach 2. Figure 5.5 **(A)** depicts the evolution of the expected weight values in synapse 0 for epoch 1 when training the memristor-based SNN directly (in blue) and the actual weights written in memristors (as orange dots). The curves are plotted every 175 samples. The expected weight (in blue) gradually approaches the optima. However, the actual weights represented by memristor conductance stop updating at a certain point because the

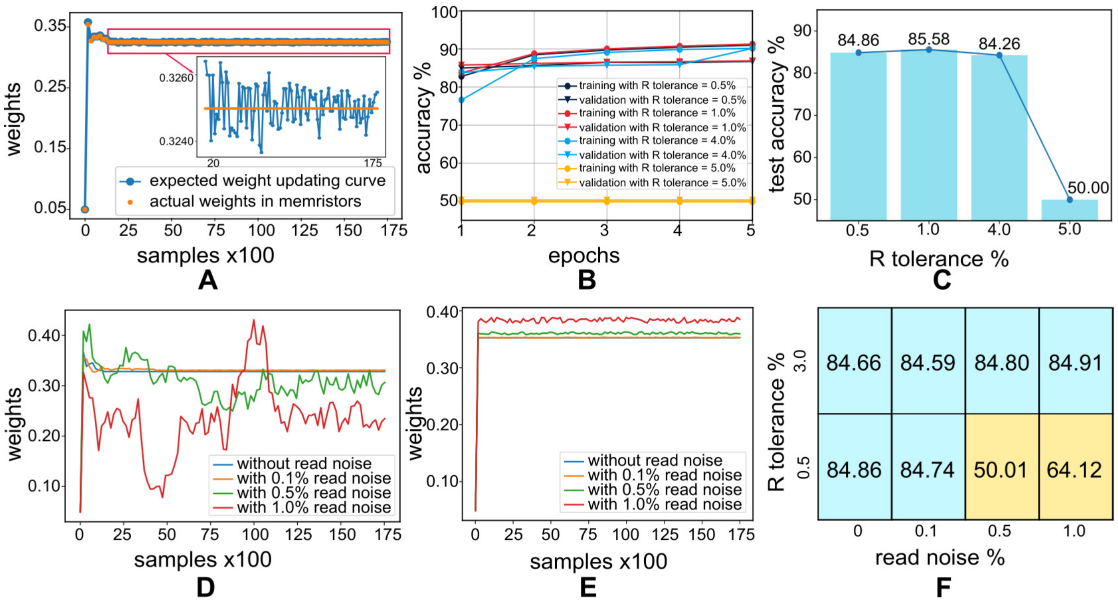


FIGURE 5.5: Result analysis for directly training the memristor-based SNN. (A) Weight evolution in synapse 0 in the single-layer SNN during training epoch 1. The curve is plotted every 175 samples. The calculated expected weights and the actual weights mapped into memristor arrays are plotted in blue and orange, respectively. Notably, memristive devices stop updating when the expected updates are below the **R tolerance**. (B)-(C) Training/validation and test accuracy with different **R tolerance** values. (D)-(E) Measured weight evolution for synapse 0 in the training epoch 1 with different read noise values when the **R tolerance** is 0.5% and 3.0%. The baseline without the read noise is the blue curve. Weights are plotted every 175 samples. (F) Shmoo plot for the test accuracy (%) with different **R tolerance** values (y axis) and read noise values (x axis). Positive and negative results are shown in blue and yellow, respectively. The image is from the author's publication [222].

weight updating is cut off by the **R tolerance** when the weight change is within the accepted region defined by the **R tolerance**. We predict that the greater the **R tolerance** value, the quicker the weight update ceases. The training/validation accuracy curves and final test accuracy with different **R tolerance** values are shown in 5.2.3 **(B)** and **(C)**. The accuracy shows no noticeable difference when the **R tolerance** is less than 4.0%. However, when the **R tolerance** is increased to 5%, the network cannot make classification predictions due to the early cut-off.

Unlike the system built by converting a trained ANN, training the memristor-based SNN is sensitive to read noise. The weight evolution in synapse 0 during training epoch 1 with different read noise values when the **R tolerance** is 0.5% is shown in Figure 5.5 **(D)**. Measured weights are plotted every 175 samples. The baseline in blue represents the ideal situation without the read noise. The baseline weight converges to an optimal state with the value of around 0.328, and the weight curve with 0.1% read noise converges to around 0.330 due to the small read-out error. However, the weights during epoch 1 fail to converge when the read noise is higher than the **R tolerance** (specifically, with 0.5% and 1.0% read noise). When the **R tolerance** is 3.0%, which is higher than all read noise values chosen in this experiment, measured weights all converge (shown in Figure 5.5 **(E)**). A possible reason is that when the read noise is higher than the **R tolerance**, the relatively large measured errors mislead the algorithm in determining the weight changing magnitudes or even directions, introduce fluctuations, and make the **R tolerance** unable to be used as an effective stop condition. In contrast, the **R tolerance** is the primary source of errors when updating weights with the **R tolerance** higher than the read noise. Therefore, the impact of the read noise can be ignored. There are two features shown in Figure 5.5 **(E)**. Firstly, the more significant the noise, the larger the fluctuations; secondly, the larger the noise, the further the measured weight convergent state is away from the baseline. However, both the fluctuations and the distances away from the baseline seem to be within acceptable ranges, with the maximum fluctuation of $\sim \pm 0.005$ and the largest distance of 0.03. The test accuracy with different read noise values and **R tolerance** is shown in 5.5 **(F)**. We conclude that when the **R tolerance** is more than the read noise, the weights converge, and similar test accuracy can be achieved with the baseline accuracy. Otherwise, the weights fail to converge, and the final test accuracy shows negative results.

5.4 Conclusion

This chapter presents a simulation framework that employs an empirical memristor model to showcase the first text classification task using memristor-based spiking neural networks in the IMDB movie reviews dataset. Two approaches are taken to acquire trained memristor-based spiking neural networks: converting trained ANNs to memristor-based SNNs and training the memristor-based SNNs directly. Specific details of estimating the critical parameters are elaborated to guide users to achieve comparable performance when taking the presented approaches. Notably, hardware specification estimation requires more information including CMOS and memristor technology. Therefore, as an algorithm-level simulation framework, this design does not provide hardware specification estimation. Given the baseline accuracy of 86.02% from the equivalent ANN, we achieved the test accuracy of 85.88% and 84.86% with just 0.14% and 1.16% degradations, respectively. Lastly, system sensitivity to SNN stochasticity and memristor non-idealities are investigated. We concluded that similar performance is achievable in simulation, from ANNs to memristor-based SNNs, and from non-memristive synapses to data-driven virtual memristive synapses. We summarise that the simulation framework can speed up the research of memristor-based SNNs in text classification tasks and overcome the imperfection of the existing learning rules and memristor non-idealities by demonstrating a sentiment analysis task.

Chapter 6

Conclusions and Future Perspectives

6.1 Summary and Conclusions

Memristor-based SNNs exhibit new approaches toward power-efficient low-cost implementations for neuromorphic applications. Two pathways, one purely hardware and one purely software, for applying memristive technologies to applications can potentially become a standard guide for researchers in this community to follow. This thesis tries to complete the whole picture of these two pathways by adding the missing puzzle pieces to enable broader possibilities for using memristor-based SNNs in varying disciplines. Three necessary puzzle pieces for completing the picture lie in hardware infrastructure, software simulators, and application theory.

In this thesis, three individual designs have been presented in hardware infrastructure, software simulations, and application theory aspects. In chapter 2, the review of the theoretical background and the historical designs utilising memristor-based SNNs have been displayed. These further confirmed the need to construct new frameworks as hardware infrastructure and software simulators and indicate the efforts required to lay theoretical foundations for exploiting the potential of incorporating memristor-based SNNs in text classification tasks.

In chapter 3, an FPGA-based digital interface for memristor array has been presented. This interface can be used to form a 64-channel control and characterisation system for memristor arrays with parallel read/write operations and high-speed data acquisition. The digital interface has been validated by resistor handling and current-voltage sweep experiments. Although the examples are for memristors, the control system built with the presented digital interface provides more general-purpose testing functionality for other emerging memory devices. The system has been compared to existing works that deliver similar functions, and it has been shown that our system can achieve comparable high data throughput with channel-level parallelism despite the low sampling rate. The system has also been proven capable of conducting advanced testing and characterisation, such as increment pulsing programming.

Chapter 4 displayed NeuroPack, a Python-based simulator with a data-driven memristor model for neuromorphic designs. Sitting at the algorithm level, NeuroPack has been designed to assist users in fast validating neuromorphic concepts with memristors. NeuroPack includes a wide range of selectable neuron, plasticity, and device models and is compatible with user-defined models to achieve flexibility and versatility. Besides, NeuroPack also monitors device behaviours and stores internal variables for result analysis. With the aid of the embedded analysis tool, users can visualise weight evolution, memristor states, membrane voltage changes, and neuron firing history. The result visualisation further allows users to explore how intimately device- and weight updating protocol-related parameters are associated with the system performance. NeuroPack also has implemented a ′predict-write-verify′ loop as the weight updating scheme to overcome the challenge of precise memristor state control due to the non-linear switching dynamics. An MNIST handwritten digit recognition task showcased how NeuroPack assists users with memristor-based neuromorphic designs.

Chapter 5 demonstrated the first text classification tasks in a memristor-based SNN. A simulation framework employing a statistic memristor model has been developed to perform this task. Two paths have been taken to obtain a trained memristor-based SNN given the absence of an efficient learning rule: one is converting a trained ANN to its equivalent memristor-based SNN, and another is training a memristor-based SNN directly. A sentiment analysis task in the IMDB movie reviews dataset took two different paths with the developed simulation framework. Specific details of estimating the critical parameters were elaborated to guide users to achieve comparable performance when utilising the presented approaches. We achieved the final test accuracy of 85.88% and 84.86% with just 0.14% and 1.16% degradations, respectively, provided the baseline accuracy of the equivalent ANN is 86.02%. We concluded that it is possible to achieve comparable accuracy in simulation, from ANNs to memristor-based SNNs, and from non-memristive synapses to data-driven virtual memristive synapses. System sensitivity to global parameters has also been investigated and illustrated when taking two different approaches. We envisage the potential of using the simulation framework with the presented two approaches to accelerate the research of incorporating memristor-based SNNs in text classification tasks.

This thesis aims to complete the whole picture of the two pathways toward memristor-based neuromorphic applications by developing three different works to add the missing puzzle pieces. The recommended way of utilising the works presented in this thesis is as follows. To begin with, researchers should gain enough knowledge of the device's characteristics. This step can be done by running necessary testing in the control and characterisation system built with the FPGA-based digital interface presented in this thesis. Next, construct the neuromorphic system from the concept and consider the details of the neural network, including the neuron model, the plasticity rule, the encoding method, the structure of the neural networks, etc. If the target application is a text classification task, the demonstration of a sentiment analysis task with two different approaches presented in chapter 5 can be referred to. Validate the conceptual design in performing the target application in an algorithm-level software simulator (e.g. NeuroPack) with extracted device parameters for a memristor behaviour model. After confirming the design functionality for the application, the design can proceed to the hardware design stage. If the goal is to implement a PCB-based neuromorphic system, the control and characterisation platform presented in chapter 3 can be utilised to map higher-level functions (e.g. matrix multiplication) with the aid of extra circuits delivering other necessary functions to complete the neuromorphic system.

Considering the limitations, for the digital interface in the memristor control system, the goal of the design is to develop an instrument that provides all necessary functions for memristor handling. Therefore, it does not have the best power consumption and speed efficiency, compared to ASIC designs. NeuroPack supports different applications and different neural network architectures. As it was developed in Numpy, it does not support GPU acceleration. Therefore, it requires long runtime to simulate a network with a large neuron size. The design in chapter 6 is bespoke for text classification. If other applications with a larger dataset, for example, machine translation, are the target tasks, the design might not work due to unsatisfying space and runtime efficiencies.

6.2 Future Perspectives towards AI Hardware

Several things can be done to extend the usage of the presented works. Firstly, the control and characterisation system can be integrated into a chip to achieve higher data processing speed. Secondly, the device volatility model, the line resistance, and the sneak path in a memristor array can be involved in the simulators to give more realistic system performance prediction. Lastly, the methods applied in the text classification demonstration can migrate to a more complex SNNs structure to achieve comparable performance to the state-of-the-art and to solve more challenging tasks. For example, building a memristor-based spiking transformer for text classification or machine translation is a worthy attempt.

In summary, from gaining device intuition to device handling, from system conceptual validation to parameter tuning, from theoretical application analysis to practical design, we believe the presented designs can leverage the potential of utilising memristor-based SNNs in different neuromorphic applications. We also believe these works will inspire researchers in the same community to approach future application-driven AI hardware designs.

Appendix A

Implementation Details of An FPGA-based Digital Interface for Memristor Arrays

A.1 DAC81416 registers reconfigurations

TABLE A.1: DAC81416 configuration registers that need to be reconfigured according to the datasheet[176].

Register name	Address	Description
SPI configuration register	03h	Default = 0AA4h; set 0A88h to activate DACs and turn off SDO
General configuration register	04h	Default = 7F00h; set 3F00h to activate the internal reference
Broadcast configuration register	05h	Default = FFFFh; set 0000h to turn off broadcast mode
DAC power-down register	09h	Default = FFFFh; set 0000h to turn off power-down mode
DAC[15:12] range register	0Ah	Default = 0000h; set AAAAh to select -10 V to +10 V range
DAC[11:8] range register	0Bh	Default = 0000h; set AAAAh to select -10 V to +10 V range
DAC[7:4] range register	0Ch	Default = 0000h; set AAAAh to select -10 V to +10 V range
DAC[3:0] range register	0Dh	Default = 0000h; set AAAAh to select -10 V to +10 V range

A.2 Instruction Set

Each instruction has eight 32-bit data. the data format is as follows:
{instr0[31 : 0], instr1[31 : 0], instr2[31 : 0], instr3[31 : 0], instr4[31 : 0], instr5[31 : 0], instr6[31 : 0], instr7[31 : 0]}

Instruction sets are shown below:

TABLE A.2: Instruction sets

instruction	opcode	description
LD VOLT	32′h00001	Load volatage data for DACs
UP DAC	32′h00002	Enable control signal to Update DAC voltages
C READ	32′h00004	Set related switches and start a current read
V READ	32′h00008	Set related switches and start a voltage read
UP SEL	32′h00010	Update selectors
UP LGC	32′h00020	Update logic
UP CH	32′h00040	Update channel switches
CLEAR	32′h00080	Set all DACs to 0V and open all switches
HS CON	32′h00100	Set the timer for high-speed pulse generation
HS PLS	32′h00200	Generate high-speed pulses
MOD CH	32′h00400	Modify control bits for switch chains
LD OFF	32′h01000	Set DAC offset registers
DELAY	32′h02000	Halt the execution for a specified time
DAC RNG	32′h04000	Set DAC ranges
HS PAT	32′h08000	Generate patterned high-speed pulses
AMP PRP	32′h10000	Connect the feedback resistors in a controlled manner

Appendix B

Implementation Details of NeuroPack

B.1 Derivation of back propagation for the leaky integrate-and-fire neuron models (without winner-take-all)

Leaky integrate-and-fire neuron model in matrix form:

$$V_t = Wx_t + \alpha V_{t-1} \odot (1 - y_{t-1})$$

$$y_t = h(V_t - V_{th})$$

$$E = \frac{1}{2N} \sum_{i=0}^{N} (y_{i,t} - \hat{y_{i,t}})^2$$

$$\Delta W = -\eta \frac{\partial E}{\partial W}$$

Where V_t is membrane voltage vector in the selected layer at time t with the size of $n \times 1$ (n is the total number of neurons in this layer), x_t is the input signals fed to the selected layer at time t with the size of $m \times 1$ (m is the total number of neurons in the last layer), W represents the weight matrix between the current current layer and the last layer with the size of $n \times m$, α is the leakage factor, $\odot$ denotes the element-wise product, V_{th} is the membrane voltage threshold, y_t is the output vector for the selected layer with only 0 and 1 to indicate if there is a spike generated with the size of $n \times 1$, and $\hat{y}_t$ is the correct output state for output layer with the same size of that of the output layer, E is the cost function calculating the distance between y_t in the output layer and $\hat{y}_t$, N is the number of output neurons, and ΔW gives the weight update matrix with the same size as that of W.

According to the chain rule, $\frac{\partial E}{\partial w_{l,t}}$ for the output layer can be derived as below:

$$\begin{aligned}\frac{\partial E}{\partial W_{l,t}} &= \frac{\partial E}{\partial y_{l,t}} \cdot \frac{\partial y_{l,t}}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l,t}} \\ &= (y_{l,t} - \hat{y}_t) \odot h'(V_{l,t} - V_{th}) \cdot x_{l,t}^T \\ &= \delta_{l,t} x_{l,t}^T \quad \textbf{let } \delta_{l,t} = (y_{l,t} - \hat{y}_t) \odot h'(V_{l,t} - V_{th})\end{aligned}$$

After that, we can back-propagate the error from the output layer:

$$\begin{aligned}\frac{\partial E}{\partial W_{l-1,t}} &= \frac{\partial E}{\partial y_{l,t}} \cdot \frac{\partial y_{l,t}}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l-1,t}} \\ &= (y_{l,t} - \hat{y}_t) \odot h'(V_{l,t} - V_{th}) \cdot \frac{\partial V_{l,t}}{\partial W_{l-1,t}} \\ &= (W_{l,t}^T \delta_{l,t} \odot h'(V_{l-1,t} - V_{th})) x_{l-1,t}^T \\ &= \delta_{l-1,t} x_{l-1,t}^T \quad \textbf{let } \delta_{l-1,t} = W_{l,t}^T \delta_{l,t} \odot h'(V_{l-1,t} - V_{th})\end{aligned}$$

Therefore, the weight update matrix ΔW can be expressed as follow:

$$\begin{aligned}\Delta W_k &= -\eta \delta_{k,t} x_{k,t}^T \\ \delta_{k,t} &= \begin{cases} \frac{1}{N}(y_{k,t} - \hat{y}_t) \odot h'(V_{k,t} - V_{th}) & \textbf{if k = K} \\ (W_{k+1,t}^T \delta_{k+1,t}) \odot h'(V_{k,t} - V_{th}) & \textbf{otherwise} \end{cases}\end{aligned}$$

Where K is the index of the output layer.

The step function $h(V_t - V_{th})$ is non-differentiable, therefore we use a straight-through estimator proposed by this work [201] shown as follow:

$$h'(V_t - V_{th}) = \begin{cases} \frac{1}{2V_{th}} & \textbf{if } 0 < V_t < 2V_{th} \\ 0 & \textbf{otherwise} \end{cases}$$

B.2 Derivation of back propagation for Leaky integrate-and-fire neuron models (with winner-take-all)

A softmax layer is added to the previous output layer to have a winner-take-all network-level restriction to only allow at most one neuron to fire at each time step:

$$S_t = softmax(V_t \odot y_t)$$

Where S_t is a vector with the size of $N \times 1$. With the new output representation becoming continuous values, we apply the cross-entropy loss as the cost function:

$$E = -\sum_{i=0}^{N} \hat{y_{i,t}} ln(S_{i,t}) = -ln(S_{j,t})$$

Where j is the index of the output neuron that should fire according to the ground truth.

Therefore, the derivation of the gradient for the output layer is as below:

$$\begin{aligned}
\frac{\partial E}{\partial W_{l,t}} &= \frac{\partial E}{\partial S_t} \cdot \frac{\partial S_t}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l,t}} \\
&= \frac{\partial E}{\partial S_t} \cdot \frac{\partial S_t}{\partial Z_{l,t}} \cdot \frac{\partial Z_{l,t}}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l,t}} \quad \textbf{let } Z_{l,t} = V_{l,t} \odot y_{l,t} \\
&= (S_t - \hat{y}_t) \odot (y_{l,t} + V_{l,t} \odot h'(V_{l,t} - V_{th}) x_{l,t}^T \\
&= \delta_{l,t} x_{l,t}^T \quad \textbf{let } \delta_{l,t} = (S_t - \hat{y}_t) \odot (y_{l,t} + V_{l,t} \odot h'(V_{l,t} - V_{th}))
\end{aligned}$$

From the output layer results, we can backpropagate the errors and get the expression for other layers as follow:

$$\begin{aligned}
\frac{\partial E}{\partial W_{l-1,t}} &= \frac{\partial E}{\partial S_t} \cdot \frac{\partial S_t}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l-1,t}} \\
&= \frac{\partial E}{\partial S_t} \cdot \frac{\partial S_t}{\partial Z_{l,t}} \cdot \frac{\partial Z_{l,t}}{\partial V_{l,t}} \cdot \frac{\partial V_{l,t}}{\partial W_{l-1,t}} \quad \textbf{let } Z_{l,t} = V_{l,t} \odot y_{l,t} \\
&= (W_{l,t}^T \delta_{l,t} \odot h'(V_{l-1,t} - V_{th})) x_{l-1,t}^T \\
&= \delta_{l-1,t} x_{l-1,t}^T \quad \textbf{let } \delta_{l,t} = (W_{l,t}^T \delta_{l,t} \odot h'(V_{l-1,t} - V_{th})
\end{aligned}$$

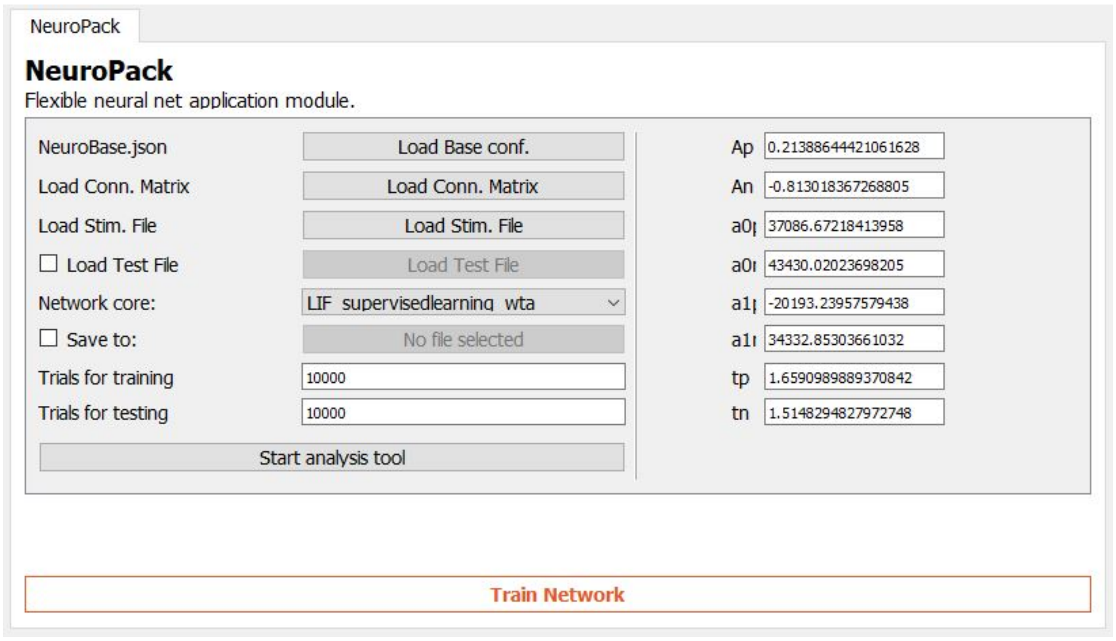
Therefore, the final weight update matrix expression is as below:

$$\begin{aligned}
\Delta W_k &= -\eta \delta_{k,t} x_{k,t}^T \\
\delta_{k,t} &= \begin{cases} (S_t - \hat{y}_t) \odot (y_t + V_t \odot h'(V_{k,t} - V_{th})) & \textbf{if k = K} \\ (W_{k+1,t}^T \delta_{k+1,t}) \odot h'(V_{k,t} - V_{th}) & \textbf{otherwise} \end{cases}
\end{aligned}$$

B.3 Pseudo code for pulse parameter selection module

```
procedure PULSE PARAMETERS SELECTION ACCORDING TO ΔW
    memristor = ParametericDevice(*args)    ▷ Create a ParametericDevice object
    res = []    ▷ Create an empty list to store resulting R
    for pulse in pulseList do    ▷ Loop over all options of pulse parameters
        memristor.initialise(R)    ▷ Set current R to R
        for timestep in range(pulse[1] / dt) do    ▷ pulse[1] stores pulsewidth
            step_dt(pulse[0], dt)    ▷ pulse[0] stores magnitude
        end for
        res.append(memristor.Rmem)    ▷ Append resulting R to the list
    end for
    resDist = abs(res - R_expected)    ▷ Calculate distance between expected R and calculated R
    selectedPulseIndex = argmin(resDist)    ▷ Find the index of minimal distance
    selectedPulse = res[selectedPulseIndex]    ▷ Find parameters of the pulse option
end procedure
```

B.4 Main Panel GUI



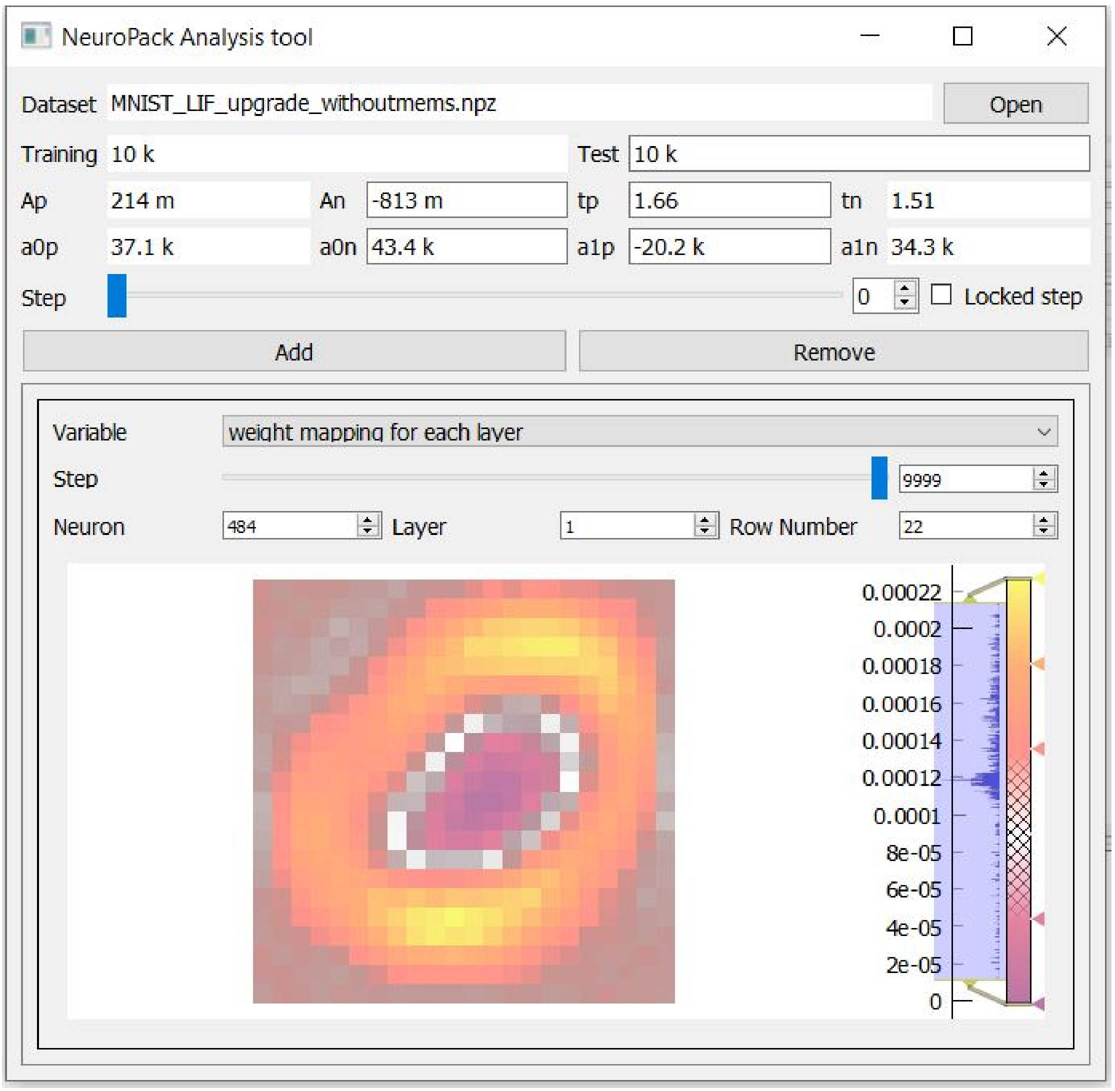
NeuroPack
Flexible neural net application module.

NeuroBase.json	Load Base conf.	Ap	0.21388644421061628
Load Conn. Matrix	Load Conn. Matrix	An	-0.813018367268805
Load Stim. File	Load Stim. File	a0p	37086.67218413958
☐ Load Test File	Load Test File	a0n	43430.02023698205
Network core:	LIF_supervisedlearning_wta	a1p	-20193.23957579438
☐ Save to:	No file selected	a1n	34332.85303661032
Trials for training	10000	tp	1.6590989889370842
Trials for testing	10000	tn	1.5148294827972748

FIGURE B.1: Screenshot of the main panel.

Figure B.1 shows the screenshot of the main panel. Users can load input files through the main panel, select the core file (the neuron model and the learning rule), and set memristor parameters. 'Network cores' gives a drop-down list showing all examples and user-defined core files. Selecting one of them sets the neuron model and the learning rule for the simulation. To execute the run, press the "Train Network" button. This action will generate a "run file" containing a log of membrane voltages, weights, etc., generated during the run and can later be opened by the analysis tool for further investigation. The 'Start analysis tool' button boots the data analysis tool.

B.5 Analysis Tool GUI


NeuroPack Analysis tool

Dataset MNIST_LIF_upgrade_withoutmems.npz

Training 10 k Test 10 k

Ap 214 m An -813 m tp 1.66 tn 1.51

a0p 37.1 k a0n 43.4 k a1p -20.2 k a1n 34.3 k

Step 0 ☐ Locked step

Variable weight mapping for each layer

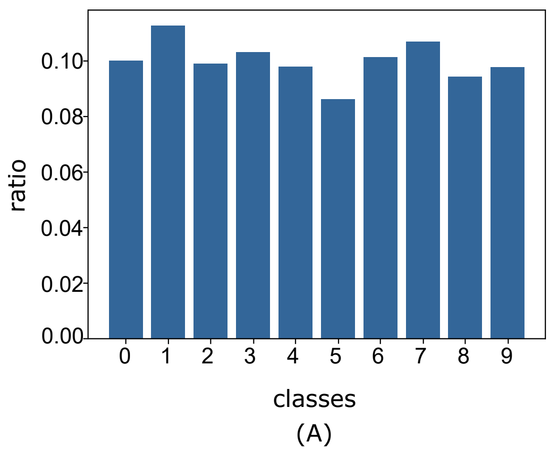
Step 9999

Neuron 484 Layer 1 Row Number 22

FIGURE B.2: Screenshot of the analysis tool.

Figure B.2 shows the screenshot of the analysis tool. It runs in a separate panel (from the main panel) but is invoked from the main panel, as shown in Figure B.1. To run the analysis tool, users need to enable ’Save to’ with a given file name to store inference and training results in the main panel. The analysis tool selects the desired ”run file” file, and parameters such as training & test epochs and memristor parameters are automatically extracted and displayed. By clicking the ’Add’ button, an analysis window is added. Users can then select a variable to plot from the drop-down list (in this example, the weight map for a particular neuron), which will display the corresponding images for a selected time step. Selectable variables include weight mappings, stimuli inputs, membrane voltages, fire history, etc. This figure shows the weight mapping for neuron 484 in layer 1 (the output layer), displayed as a rectangle with 22 pixels in a row. A weight distribution summary is also displayed to the right of the image (weights are shown in units of Siemens in the current implementation).

B.6 Dataset



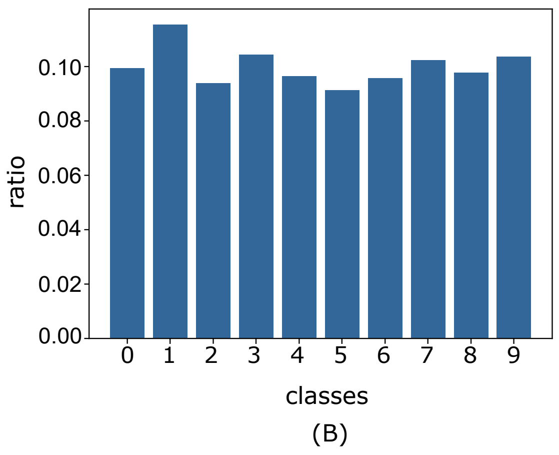


FIGURE B.3: Proportions of digits in the whole MINST training set (A - 10k samples), and our chosen test set (B - reduced to 2k samples)

The calculated euclidean distance between the used 2k and the original 10k test sets is 0.019 using the equation below:

$$\text{Euclidean distance} = \sqrt{\sum (f_{\text{original MNIST}} - f_{\text{2k test set}})^2}$$

Where f represents the statistic frequency of a digit in the whole dataset.

B.7 Accuracy improvement

We investigated why the baseline accuracy given in the 'Results' part is relatively low. We tried three mitigation methods: increasing test data samples, training samples, and fine-tuning parameters. Figure B.4 summarises what we found:

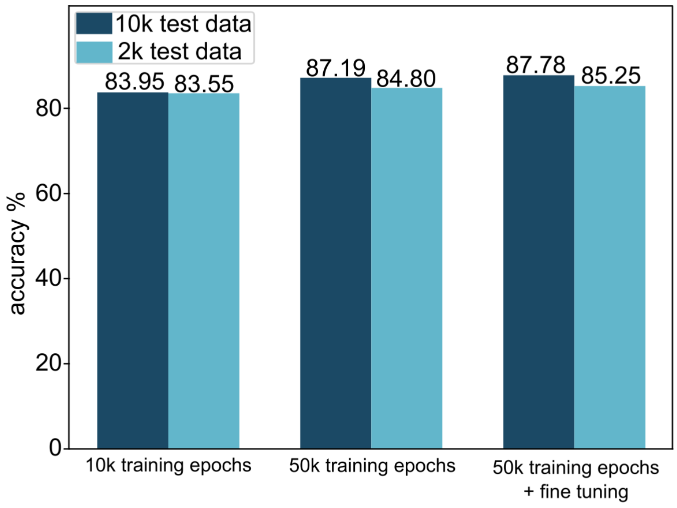


FIGURE B.4: Accuracy comparison between using 10k test data and 2k test data for different methods.

We found a slight difference between 10k test data and 2k test data. We attribute this to the fact that the 2k test dataset is relatively small, and the slight unbalance of test data creates the difference in test accuracy for each class (shown in Figure B.5) to the total test accuracy.

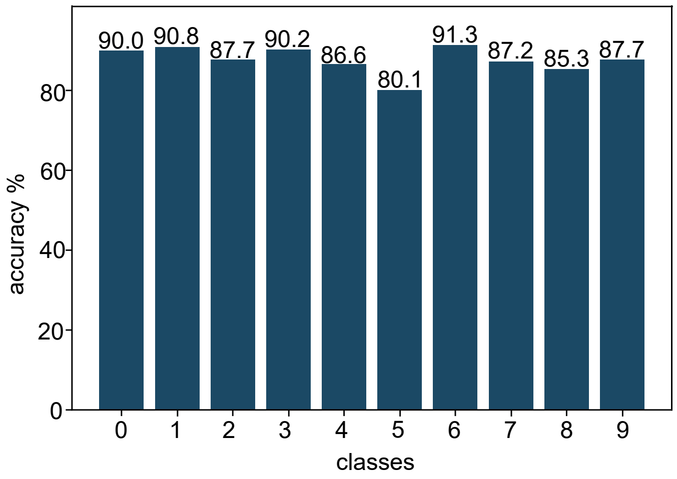


FIGURE B.5: Test accuracy for each class.

We also found that training for longer epochs and fine-tuning parameters help bring $\sim 4\%$ improvement in accuracy. Figure B.6 shows the training accuracy evolution for the versions that give test accuracy of 87.19% (before parameter fine-tuning) and 87.78% (after parameter fine-tuning).

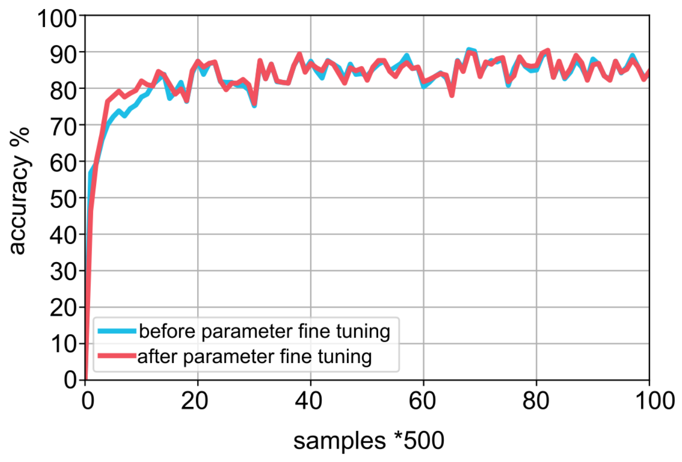


FIGURE B.6: Training accuracy for running 50k epochs. The figure is plotted every 500 samples for clarity.

Appendix C

Mathematical Derivation for Text Classification in Memristor-based Spiking Neural Networks

C.1 Relation between the memristor-based SNN firing rates and the equivalent-ANN outputs

Inspired by the derivation given by this work [225], now we derive the relation between the memristor-based SNN output firing rates and the equivalent-ANN outputs.

Assume we have an SNN with an integrate-and-fire neuron model described as follows:

$$V_t = V_{t-1} + \sum W_s x_t - Z_{t-1} V_{th} \tag{C.1}$$

$$Z_t = h(V_t - V_{th}) \tag{C.2}$$

Where V_t, x_t, and Z_t denote the membrane voltages, input spikes, and output spikes at time step t, V_{th} is the membrane voltage threshold, W_s represents the weights in the single-layer SNN, and $h(x)$ is the Heaviside step function. We also have its equivalent ANN described using the equation below:

$$V_c = \sum W_s x_c \tag{C.3}$$

Where V_c and x_c are the continuous-valued ANN outputs and inputs. x_t is the spike train representation of x_c, and their relation can be expressed using the equation below:

$$x_c = \frac{1}{T}\sum^{T} x_t \tag{C.4}$$

Where T is the length of the spike train to represent a single continuous value.

Now we accumulate the membrane voltage V_t over the time window T using Equation C.1, rearrange it and use Equation C.3 and C.4 to simplify it, we can write the expression of the output firing state Z_{t-1}:

$$\begin{aligned}
\sum^{T} V_t &= \sum^{T} V_{t-1} + \sum^{T}\sum W_s x_t - \sum^{T} Z_{t-1} V_{th} \\
V_t - V_0 &= \sum W_s \sum^{T} x_t - V_{th} \sum^{T} Z_{t-1} \\
V_t - V_0 &= T \sum W_s x_c - V_{th} \sum^{T} Z_{t-1} \\
V_t - V_0 &= T \cdot V_c - V_{th} \sum^{T} Z_{t-1} \\
\sum^{T} Z_{t-1} &= \frac{T \cdot V_c}{V_{th}} - \frac{V_t - V_0}{V_{th}}
\end{aligned}$$

By dividing by T on both sides, we can get the approximation of the firing rate as follow:

$$r = \frac{\sum^{T} Z_t}{T} \approx \frac{\sum^{T} Z_{t-1}}{T} = \frac{V_c}{V_{th}} - \frac{V_t - V_0}{V_{th} \cdot T} \tag{C.5}$$

Where V_0 denotes the initial membrane voltage of a spiking neuron when starting a new inference. From the equation, we can see that the memristor-based SNN output firing rates are proportional to the equivalent ANN continuous-valued outputs V_c, with a constant coefficient V_c/V_{th} and an error term $(V_t - V_0)/(V_{th} \cdot T)$.

References

[1] N. Sengupta, M. Sahidullah, and G. Saha, "Lung sound classification using cepstral-based statistical features," *Computers in Biology and Medicine*, vol. 75, pp. 118–129, 2016.

[2] C. B. Choy, D. Xu, J. Gwak, K. Chen, and S. Savarese, "3d-r2n2: A unified approach for single and multi-view 3d object reconstruction," *CoRR*, vol. abs/1604.00449, 2016. arXiv:1604.00449.

[3] D. S. Maitra, U. Bhattacharya, and S. K. Parui, "Cnn based common approach to handwritten character recognition of multiple scripts," in *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pp. 1021–1025, 2015.

[4] H. Lu, R. Setiono, and H. Liu, "Effective data mining using neural networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 957–961, 1996.

[5] M. W. Craven and J. W. Shavlik, "Using neural networks for data mining," *Future Generation Computer Systems*, vol. 13, no. 2, pp. 211–229, 1997. Data Mining.

[6] D. Zissis, E. K. Xidias, and D. Lekkas, "A cloud based architecture capable of perceiving and predicting multiple vessel behaviour," Oct. 2015.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[8] N. Ganesan, K. Venkatesh, M. Rama, and A. M. Palani, “Application of neural networks in diagnosing cancer disease using demographic data,” *International Journal of Computer Applications*, vol. 1, no. 26, pp. 76–85, 2010.

[9] L. Bottaci, P. J. Drew, J. E. Hartley, M. B. Hadfield, R. Farouk, P. W. R. Lee, I. M. Macintyre, G. S. Duthie, and J. Monson, “Artificial neural networks applied to outcome prediction for colorectal cancer patients in separate institutions,” *The Lancet*, vol. 350, pp. 469–472, 1997.

[10] Y. Li and W. Ma, “Applications of artificial neural networks in financial economics: A survey,” in *2010 International Symposium on Computational Intelligence and Design*, vol. 1, pp. 211–214, 2010.

[11] R. Dase and D. Pawar, “Application of artificial neural network for stock market predictions: A review of literature,” *International Journal of Machine Intelligence*, vol. 2, 12 2010.

[12] S. P. Singh, A. Kumar, H. Darbari, L. Singh, A. Rastogi, and S. Jain, “Machine translation using deep learning: An overview,” in *2017 International Conference on Computer, Communications and Electronics (Comptelix)*, pp. 162–167, 2017.

[13] W. Maass, “Networks of spiking neurons: The third generation of neural network models,” *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[14] J. Vreeken, “Spiking neural networks, an introduction,” 2003.

[15] H. Takagi, “Roles of ion channels in epsp integration at neuronal dendrites,” *Neuroscience Research*, vol. 37, no. 3, pp. 167–171, 2000.

[16] J. S. Coombs, J. C. Eccles, and P. Fatt, “The inhibitory suppression of reflex discharges from motoneurones,” *The Journal of physiology*, vol. 130, no. 2, pp. 396–413, 1955.

[17] A. Khodamoradi, K. Denolf, and R. Kastner, “S2n2: A fpga accelerator for streaming spiking neural networks,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 194–205, 2021.

[18] K. Ahmed, A. Shrestha, Q. Qiu, and Q. Wu, “Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor,” in *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 4286–4293, 2016.

[19] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Neuron fault tolerance in spiking neural networks," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 743–748, 2021.

[20] U. Ramacher, J. Beichter, and N. Bruls, "Architecture of a general-purpose neural signal processor," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. i, pp. 443–446 vol.1, 1991.

[21] D. Hammerstrom, "A vlsi architecture for high-performance, low-cost, on-chip learning," in *1990 IJCNN International Joint Conference on Neural Networks*, pp. 537–544 vol.2, 1990.

[22] K. Mohraz, U. Schott, and M. Pauly, "Parallel simulation of pulse coded neural networks," 1997.

[23] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, "Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," in *Euro-Par 2007 Parallel Processing*, (Berlin, Heidelberg), pp. 672–681, Springer Berlin Heidelberg, 2007.

[24] E. Niebur and D. Brettle, "Efficient simulation of biological neural networks on massively parallel supercomputers with hypercube architecture," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, p. 904–910, Morgan Kaufmann Publishers Inc., 1993.

[25] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," in *2009 International Joint Conference on Neural Networks*, pp. 2145–2152, 2009.

[26] A. Jahnke, T. Schönauer, U. Roth, K. Mohraz, and H. Klar, "Simulation of spiking neural networks on different hardware platforms," in *Artificial Neural Networks — ICANN'97*, (Berlin, Heidelberg), pp. 1187–1192, Springer Berlin Heidelberg, 1997.

[27] U. Roth, A. Jahnke, and H. Klar, "Hardware requirements for spike-processing neural networks," in *From Natural to Artificial Neural Computation*, (Berlin, Heidelberg), pp. 720–727, Springer Berlin Heidelberg, 1995.

[28] J. Backus, “Can programming be liberated from the von neumann style? a functional style and its algebra of programs,” *Commun. ACM*, vol. 21, p. 613–641, Aug. 1978.

[29] N. R. Mahapatra and B. Venkatrao, “The processor-memory bottleneck: Problems and solutions,” *XRDS*, vol. 5, p. 2–es, Apr. 1999.

[30] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[31] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, *et al.*, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, 2014.

[32] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, *et al.*, “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, 2018.

[33] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, *et al.*, “Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations,” *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[34] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, “A wafer-scale neuromorphic hardware system for large-scale neural modeling,” *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 1947–1950, 2010.

[35] S. Furber, F. Galluppi, S. Temple, and L. Plana, “The spinnaker project,” *Proceedings of the IEEE*, vol. 102, pp. 652–665, 05 2014.

[36] D. Thomas and W. Luk, “Fpga accelerated simulation of biologically plausible spiking neural networks,” in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 45–52, 2009.

[37] K. Cheung, S. R. Schultz, and W. Luk, “A large-scale spiking neural network accelerator for fpga systems,” in *Artificial Neural Networks and Machine Learning – ICANN 2012*, (Berlin, Heidelberg), pp. 113–120, Springer Berlin Heidelberg, 2012.

[38] D. Neil and S. C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[39] H. Mostafa, B. U. Pedroni, S. Sheik, and G. Cauwenberghs, "Fast classification using sparsely active spiking networks," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, 2017.

[40] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu, "Encoding, Model, and Architecture: Systematic Optimization for Spiking Neural Network in FPGAs," *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, vol. 2020-Novem, 2020.

[41] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga," 2018.

[42] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," 2015.

[43] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 26–35, ACM, 2016.

[44] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

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

[46] A. Prost-Boucle, A. BOURGE, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, (Gent, Belgium), Sept. 2017.

[47] W. Zhang, B. Gao, J. Tang, P. Yao, S. Yu, M.-F. Chang, H.-J. Yoo, H. Qian, and H. Wu, "Neuro-inspired computing chips," *Nature Electronics*, 07 2020.

[48] A. Sebastian, M. Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, 03 2020.

[49] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6t sram array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, 2017.

[50] N. Verma, H. Jia, H. Valavi, Y. Tang, M. Ozatay, L. Chen, B. Zhang, and P. Deaville, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.

[51] H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma, "A 64-tile 2.4-mb in-memory-computing cnn accelerator employing charge-domain compute," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 6, pp. 1789–1799, 2019.

[52] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, *et al.*, "In-datacenter performance analysis of a tensor processing unit," 2017.

[53] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1 – 4, 10 2011.

[54] J. Seo, B. Brezzo, Y. Liu, B. D. Parker, S. K. Esser, R. K. Montoye, B. Rajendran, J. A. Tierno, L. Chang, D. S. Modha, and D. J. Friedman, "A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, 2011.

[55] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6t sram array," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 915–924, Apr. 2017.

[56] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[57] M. Hu, C. E. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, Q. Xia, and J. P. Strachan, "Memristor-based analog computation and neural network classification with a dot product engine," *Advanced Materials*, 2018.

[58] A. Serb, E. Manino, I. Messaris, L. Tran-Thanh, and T. Prodromakis, "Hardware-level bayesian inference," in *Neural Information Processing Systems*, December 2017.

[59] Z. Wang, C. Li, P. Lin, M. Rao, Y. Nie, W. Song, Q. Qiu, Y. Li, P. Yan, J. W. Strachan, N. Ge, N. McDonald, Q. wu, M. Hu, H. Wu, S. Williams, Q. Xia, and J. J. Yang, "In situ training of feed-forward and recurrent convolutional memristor networks," *Nature Machine Intelligence*, 09 2019.

[60] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, 2020.

[61] C. Li, Z. Wang, M. Rao, D. Belkin, W. Song, H. Jiang, P. Yan, *et al.*, "Long short-term memory networks in memristor crossbar arrays," *Nature Machine Intelligence*, vol. 1, 01 2019.

[62] C. Li, D. Belkin, Y. Li, P. Yan, M. Hu, N. Ge, H. Jiang, *et al.*, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature Communications*, vol. 9, 06 2018.

[63] S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, p. 60—67, June 2018.

[64] S. Stathopoulos, A. Khiat, M. Trapatseli, S. Cortese, A. Serb, I. Valov, and T. Prodromakis, "Multibit memory operation of metal-oxide bi-layer memristors," *Scientific Reports*, vol. 7, 12 2017.

[65] B. J. Choi, A. Torrezan, J. W. Strachan, P. Kotula, A. Lohn, M. Marinella, Z. Li, S. Williams, and J. J. Yang, "High-speed and low-energy nitride memristors," *Advanced Functional Materials*, vol. 26, 05 2016.

[66] A. Sebastian, M. Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, vol. 15, 03 2020.

[67] G. Adam, A. Khiat, and T. Prodromakis, "Challenges hindering memristive neuromorphic hardware from going mainstream," *Nature Communications*, vol. 9, 12 2018.

[68] G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, *et al.*, "Neuromorphic computing using non-volatile memory," *Advances in Physics: X*, vol. 2, no. 1, pp. 89–124, 2017.

[69] S. Woźniak, A. Pantazi, T. Bohnstingl, and E. Eleftheriou, "Deep learning incorporating biologically inspired neural dynamics and in-memory computing," *Nature Machine Intelligence*, vol. 2, p. 325–336, Jun 2020.

[70] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps," *Science (New York, N.Y.)*, vol. 275, pp. 213–5, 02 1997.

[71] A. Serb, J. Bill, A. Khiat, R. Berdan, R. Legenstein, and T. Prodromakis, "Unsupervised learning in probabilistic neural networks with multi-state metal-oxide memristive synapses," *Nature Communications*, 2016.

[72] Y. Guo, H. Wu, B. Gao, and H. Qian, "Unsupervised learning on resistive memory array based spiking neural networks," *Frontiers in Neuroscience*, vol. 13, no. JUL, pp. 1–16, 2019.

[73] M. Payvand, J. Rofeh, A. Sodhi, and L. Theogarajan, "A cmos-memristive self-learning neural network for pattern classification applications," in *2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 92–97, 2014.

[74] I. Boybat, M. Le Gallo, S. R. Nandakumar, T. Moraitis, T. Parnell, T. Tuma, B. Rajendran, Y. Leblebici, A. Sebastian, and E. Eleftheriou, "Neuromorphic computing with multi-memristive synapses," *Nature Communications*, vol. 9, no. 1, pp. 1–15, 2018.

[75] B. Liu, Y. Chen, B. Wysocki, and T. Huang, "Reconfigurable neuromorphic computing system with memristor-based synapse design," *Neural Processing Letters*, vol. 41, 04 2013.

[76] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu, and H. Jiang, "A spiking neuromorphic design with resistive crossbar," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[77] Y. Demirag, C. Frenkel, M. Payvand, and G. Indiveri, "Online Training of Spiking Recurrent Neural Networks with Phase-Change Memory Synapses," 2021. arXiv:1410.5093.

[78] J. Shen, A. Mifsud, L. Xie, A. Alshaya, and C. Papavassiliou, "A high-voltage characterisation platform for emerging resistive switching technologies," 2022. arXiv:2205.08391.

[79] A. Mifsud, J. Shen, P. Feng, L. Xie, C. Wang, Y. Pan, S. Maheshwari, S. Agwa, S. Stathopoulos, S. Wang, A. Serb, C. Papavassiliou, T. Prodromakis, and T. G. Constandinou, "A cmos-based characterisation platform for emerging rram technologies," 2022. arXiv:2205.08379.

[80] R. Berdan, A. Serb, A. Khiat, A. Regoutz, C. Papavassiliou, and T. Prodromakis, "A μ-controller-based system for interfacing selectorless rram crossbar arrays," *IEEE Transactions on Electron Devices*, vol. 62, no. 7, pp. 2190–2196, 2015.

[81] P.-Y. Chen, X. Peng, and S. Yu, "Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 12, pp. 3067–3080, 2018.

[82] L. Xia, B. Li, T. Tang, P. Gu, P.-Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "Mnsim: Simulation platform for memristor-based neuromorphic computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1009–1022, 2018.

[83] M. Payvand, M. E. Fouda, F. Kurdahi, A. M. Eltawil, S. Member, and E. O. Neftci, "On-Chip Error-Triggered Learning of Multi-Layer Memristive Spiking Neural Networks," vol. 10, no. 4, pp. 522–535, 2020.

[84] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[85] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. arXiv:1301.3781.

[86] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein, and W. Maass, "A solution to the learning dilemma for recurrent networks of spiking neurons," *Nature Communications*, vol. 11, 07 2020.

[87] S. Ruder, "An overview of gradient descent optimization algorithms," 2016. arXiv:1609.04747.

[88] Y. Messaris, A. Serb, A. Khiat, S. Nikolaidis, and T. Prodromakis, "A compact verilog-a reram switching model," 03 2017.

[89] L. Abbott, "Lapicque's introduction of the integrate-and-fire model neuron (1907)," *Brain Research Bulletin*, vol. 50, no. 5, pp. 303–304, 1999.

[90] M. Oster, R. J. Douglas, and S.-C. Liu, "Computation with spikes in a winner-take-all network," *Neural Computation*, vol. 21, pp. 2437–2465, 2009.

[91] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[92] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, (Portland, Oregon, USA), pp. 142–150, Association for Computational Linguistics, June 2011.

[93] M. Oster, R. Douglas, and S.-C. Liu, "Computation with spikes in a winner-take-all network," *Neural computation*, vol. 21, pp. 2437–65, 07 2009.

[94] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, pp. 2531–2560, 2002.

[95] H. Paugam-Moisy and S. Bohte, "Computing with spiking neuron networks," *Handbook of Natural Computing*, vol. 1-4, pp. 335–376, 2012.

[96] A. Hodgkin and A. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Bulletin of Mathematical Biology*, vol. 52, no. 1, pp. 25–71, 1990.

[97] E. M. Izhikevich, "Which model to use for cortical spiking neurons?," *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[98] M. J. Skocik and L. N. Long, “On the capabilities and computational costs of neuron models,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 8, pp. 1474–1483, 2014.

[99] R. STEIN, “A theoretical analysis of neuronal variability,” *Biophysical journal*, vol. 5, p. 173—194, March 1965.

[100] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. USA: Cambridge University Press, 2014.

[101] B. Lundstrom, M. Higgs, W. Spain, and A. Fairhall, “Fractional differentiation by neocortical pyramidal neurons,” *Nature neuroscience*, vol. 11, pp. 1335–42, 11 2008.

[102] W. Teka, T. Marinov, and F. Santamaria, “Neuronal spike timing adaptation described with a fractional leaky integrate-and-fire model,” *PLoS computational biology*, vol. 10, p. e1003526, 03 2014.

[103] N. Fourcaud-Trocmé, D. Hansel, C. van Vreeswijk, and N. Brunel, “How spike generation mechanisms determine the neuronal response to fluctuating inputs,” *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 23, pp. 11628–40, 01 2004.

[104] R. Brette and W. Gerstner, “Adaptive exponential integrate-and-fire model as an effective description of neuronal activity,” *Journal of neurophysiology*, vol. 94, pp. 3637–42, 12 2005.

[105] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[106] W. Gerstner, “Time structure of the activity in neural network models,” *Phys. Rev. E*, vol. 51, pp. 738–758, Jan 1995.

[107] D. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.

[108] “Hebb, d. o. the organization of behavior: A neuropsychological theory. new york: John wiley and sons, inc., 1949. 335 p. $4.00,” *Science Education*, vol. 34, no. 5, pp. 336–337, 1950.

[109] J. Sjöström and W. Gerstner, “Spike-timing dependent plasticity,” *Scholarpedia*, 2010.

[110] S. Davidson and S. Furber, "Comparison of artificial and spiking neural networks on digital hardware," *Frontiers in Neuroscience*, vol. 15, p. 651141, 04 2021.

[111] B. J. Choi, A. Torrezan, J. W. Strachan, P. Kotula, A. Lohn, M. Marinella, Z. Li, S. Williams, and J. J. Yang, "High-speed and low-energy nitride memristors," *Advanced Functional Materials*, vol. 26, 05 2016.

[112] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, p. 80—83, May 2008.

[113] J. P. Strachan, A. C. Torrezan, G. Medeiros-Ribeiro, and R. S. Williams, "Measuring the switching dynamics and energy efficiency of tantalum oxide memristors," *Nanotechnology*, vol. 22, p. 505402, nov 2011.

[114] R. Waser, R. Dittmann, G. Staikov, and K. Szot, "Redox-based resistive switching memories – nanoionic mechanisms, prospects, and challenges," *Advanced Materials*, vol. 21, no. 25-26, pp. 2632–2663, 2009.

[115] N. P. Jouppi, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 1–12, 2017.

[116] S. K. Gonugondla, M. Kang, and N. Shanbhag, "A 42pj/decision 3.12tops/w robust in-memory machine learning classifier with on-chip training," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 490–492, 2018.

[117] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, 2020.

[118] S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, pp. 60–67, jun 2018.

[119] G. Indiveri, E. Chicca, and R. Douglas, "A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE Transactions on Neural Networks*, vol. 17, no. 1, pp. 211–221, 2006.

[120] T. Serrano-Gotarredona, T. Masquelier, T. Prodromakis, G. Indiveri, and B. Linares-Barranco, "Stdp and stdp variations with memristors forspiking neuromorphic learning systems," *Frontiers in Neuroscience*, vol. 7, pp. 1–15, February 2013.

[121] S. Yu, P. Chen, Y. Cao, L. Xia, Y. Wang, and H. Wu, "Scaling-up resistive synaptic arrays for neuro-inspired architecture: Challenges and prospect," in *2015 IEEE International Electron Devices Meeting (IEDM)*, pp. 17.3.1–17.3.4, 2015.

[122] L. Gao, I.-T. Wang, P.-Y. Chen, S. Vrudhula, J.-s. Seo, Y. Cao, T.-H. Hou, and S. Yu, "Fully parallel write/read in resistive synaptic array for accelerating on-chip learning," *Nanotechnology*, vol. 26, p. 455204, 10 2015.

[123] Z. Bing, C. Meschede, F. Röhrbein, K. Huang, and A. C. Knoll, "A survey of robotics control based on learning-inspired spiking neural networks," *Frontiers in Neurorobotics*, vol. 12, p. 35, 2018.

[124] M. R. Azghadi, C. Lammie, J. K. Eshraghian, M. Payvand, E. Donati, B. Linares-Barranco, and G. Indiveri, "Hardware implementation of deep network accelerators towards healthcare and biomedical applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 14, no. 6, pp. 1138–1159, 2020.

[125] E. Ceolini, C. Frenkel, S. B. Shrestha, G. Taverni, L. Khacef, M. Payvand, and E. Donati, "Hand-gesture recognition based on emg and event-based camera sensor fusion: A benchmark in neuromorphic computing," *Frontiers in Neuroscience*, vol. 14, p. 637, 2020.

[126] S. Yin, S. K. Venkataramanaiah, G. K. Chen, R. Krishnamurthy, Y. Cao, C. Chakrabarti, and J. Seo, "Algorithm and hardware design of discrete-time spiking neural networks based on back propagation with binary activations," *CoRR*, vol. abs/1709.06206, 2017.

[127] N. Zheng and P. Mazumder, "A low-power hardware architecture for online supervised learning in multi-layer spiking neural networks," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2018.

[128] P. Lenander and A. Fosselius, "A survey of the spinnaker project : A massively parallel spiking neural network architecture," 2010. https://www.semanticscholar.org/paper/A-survey-of-the-SpiNNaker-Project-%3A-A-massively-Lenander-Fosselius/72f3f039cbf6cc1958ead450acf7917e3c9d3800.

[129] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation," *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012*, pp. 133–140, 2012.

[130] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, p. 1–35, Jun 2019.

[131] S. Scholze, H. Eisenreich, S. Höppner, G. Ellguth, S. Henker, M. Ander, S. Hänzsche, J. Partzsch, C. Mayr, and R. Schüffny, "A 32 gbit/s communication soc for a waferscale neuromorphic system," *Integration the VLSI Journal*, vol. 45, pp. 61–75, 01 2012.

[132] S. Narayanan, A. Shafiee, and R. Balasubramonian, "INXS: Bridging the throughput and energy gap for spiking neural networks," *Proceedings of the International Joint Conference on Neural Networks*, vol. 2017-May, pp. 2451–2459, 2017.

[133] S. Furber, "Large-scale neuromorphic computing systems," 2016. http://stacks.iop.org/1741-2552/13/i=5/a=051001?key=crossref.c170009b6af4be582bc1fbfbf3ccad04.

[134] C. Lammie, W. Xiang, and M. Rahimi Azghadi, "Modeling and simulating in-memory memristive deep learning systems: An overview of current efforts," *Array*, vol. 13, p. 100116, 2022.

[135] D. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in python," *Frontiers in Neuroinformatics*, vol. 2, 2008.

[136] M. Stimberg, D. Goodman, and T. Nowotny, "Brian2genn: accelerating spiking neural network simulations with graphics hardware," *Scientific Reports*, vol. 10, 01 2020.

[137] M. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007. revision #130182.

[138] M. L. Hines and N. T. Carnevale, "Neuron: A tool for neuroscientists," *The Neuroscientist*, vol. 7, no. 2, pp. 123–135, 2001. PMID: 11496923.

[139] J. Vitay, H. Dinkelbach, and F. Hamker, "Annarchy: a code generation approach to neural simulations on parallel hardware," *Frontiers in Neuroinformatics*, vol. 9, 2015.

[140] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. Stewart, D. Rasmussen, X. Choo, A. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, vol. 7, 2014.

[141] N. K. Kasabov, "Neucube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data," *Neural Networks*, vol. 52, pp. 62–76, 2014.

[142] H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, "Bindsnet: A machine learning-oriented spiking neural networks library in python," *Frontiers in Neuroinformatics*, vol. 12, 2018.

[143] M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, and T. Masquelier, "Spyketorch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron," *Frontiers in Neuroscience*, vol. 13, 2019.

[144] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," *CoRR*, vol. abs/2109.12894, 2021.

[145] W. Fei, H. Yu, W. Zhang, and K. S. Yeo, "Design exploration of hybrid cmos and memristor circuit by new modified nodal analysis," 2011.

[146] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[147] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, pp. 392–397, 2012.

[148] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Comput. Archit. Lett.*, vol. 14, p. 140–143, jul 2015.

[149] S. Roy, S. Sridharan, S. Jain, and A. Raghunathan, "Txsim: Modeling training of deep neural networks on resistive crossbar systems," *CoRR*, vol. abs/2002.11151, 2020.

[150] C. Lammie, W. Xiang, B. Linares-Barranco, and M. R. Azghadi, "Memtorch: An open-source simulation framework for memristive deep learning systems," *CoRR*, vol. abs/2004.10971, 2020.

[151] M. J. Rasch, D. Moreda, T. Gokmen, M. Le Gallo, F. Carta, C. Goldberg, K. El Maghraoui, A. Sebastian, and V. Narayanan, "A flexible and fast pytorch toolkit for simulating training and inference on analog crossbar arrays," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 1–4, 2021.

[152] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *eLife*, vol. 8, p. e47314, aug 2019.

[153] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[154] L. Nagel and D. O. Pederson, "Spice (simulation program with integrated circuit emphasis)," 1973.

[155] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[156] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*, 1999.

[157] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[158] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[159] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (San Diego, California), pp. 1480–1489, Association for Computational Linguistics, June 2016.

[160] X. Feng, H. Zhang, Y. Ren, P. Shang, Y. Zhu, Y. Liang, R. Guan, and D. Xu, "Pubmender: Deep learning based recommender system for biomedical publication venue (preprint)," *Journal of Medical Internet Research*, vol. 21, 11 2018.

[161] D. Chicco, P. Sadowski, and P. Baldi, "Deep autoencoder neural networks for gene ontology annotation predictions," BCB '14, (New York, NY, USA), p. 533–540, Association for Computing Machinery, 2014.

[162] S. Woźniak, A. Pantazi, T. Bohnstingl, and E. Eleftheriou, "Deep learning incorporating biologically inspired neural dynamics and in-memory computing," *Nature Machine Intelligence*, vol. 2, no. 6, pp. 325–336, 2020.

[163] G. S. Snider, "Architecture and methods for computing with reconfigurable resistor crossbars," U.S. Patent 7203789B2, April, 2007.

[164] B. L. Mouttet, "Crossbar control circuit," U.S. Patent 7609086B2, Oct, 2009.

[165] H. B. Kang, "Rfid device with memory unit having memristor characteristics," U.S. Patent 8113437B2, June, 2011.

[166] B. L. Mouttet, "Programmable crossbar signal processor," U.S. Patent 7302513B2, Nov, 2011.

[167] B. L. Mouttet, "Memristor crossbar neural interface," U.S. Patent 7902867B2, March, 2011.

[168] A. Serb, E. Manino, I. Messaris, L. Tran-Thanh, and T. Prodromakis, "Hardware-level bayesian inference," in *Neural Information Processing Systems*, December 2017.

[169] P. Foster, J. Huang, A. Serb, S. Stathopoulos, C. Papavassiliou, and T. Prodromakis, "An fpga-based system for generalised electron devices testing," 2022. ArXiv:2202.00499.

[170] P. Foster, J. Huang, A. Serb, T. Prodromakis, and C. Papavassiliou, "An fpga based system for interfacing with crossbar arrays," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, 2020.

[171] P. Foster, J. Huang, A. Serb, T. Prodromakis, and C. Papavassiliou, "Live demonstration: Electroforming of tio$_{2-x}$ memristor devices using high speed pulses," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–1, 2020.

[172] Cesys, "AXI-FX3-Interface v1.2," 2017. https://www.cesys.com/fileadmin/user_upload/service/FPGA/fpga%20 boards%20%26%20modules/EFM-03/ug121-axi-fx3-interface-1.2.1.pdf.

[173] Xilinx Inc., "PG057 FIFO Generator v13.1," p. 218, 2017. https://www.xilinx.com/support/documentation/ip_documentation/ fifo_generator/v13_1/pg057-fifo-generator.pdf.

[174] Xilinx Inc., "7 Series FPGAs Memory Resources: User Guide," *Xilinx Technical Documentation*, vol. 1.14, no. ug473, pp. 1–88, 2019. https://www.xilinx.com/support/documentation/user_guides/ug473_ 7Series_Memory_Resources.pdf.

[175] Cesys, "Efm-03 hardware reference," 2018. https://www.cesys.com/fileadmin/user_upload/service/FPGA/fpga% 20boards%20%26%20modules/EFM-03/ug-120-EFM-03-hardware-reference.pdf.

[176] Texas Instruments, "Dacx1416 16-channel, 16-,14-,12-bit, high-voltage output dacs with internal reference," 2018. http://www.ti.com/lit/ds/symlink/dac81416.pdf.

[177] Analog Devices, "Buffered octal, 16-bit, 200ksps/ch differential ±10.24v adc with 30vp-p common mode range," 2018. https://www.analog.com/media/en/technical-documentation/data-sheets/ltc2358-16.pdf.

[178] Analog Devices, "1% R-Tolerance Digital Potentiometer," 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/AD5293.pdf.

[179] ARM, *AMBA® AXI™ and ACE™ Protocol Specification*, 2013. https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf.

[180] Xilinx Inc., "AXI Interconnect v2.1 LogiCORE IP Product GuideAXI Interconnect v2.1 LogiCORE IP Product Guide," *Xilinx Technical Documentation*, no. PG059, 2022. https://docs.xilinx.com/r/en-US/pg059-axi-interconnect/AXI-Data-Width-Converter?tocId=zWQGVMqxoe4OuTrowyi9Mg.

[181] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007. isbn: 0470054379.

[182] A. Yesil, F. Gül, and Y. Babacan, *Emulator Circuits and Resistive Switching Parameters of Memristor*, pp. 41–61. 04 2018.

[183] DIGILENT, *Analog Discovery 2™ Reference manual*, 2015. https://digilent.com/reference/_media/reference/instrumentation/analog-discovery-2/ad2_rm.pdf.

[184] Arc Instruments, *Memristor Characterisation Platform User manual*, 2017. http://files.arc-instruments.co.uk/documents/ArC_One.pdf.

[185] Keithley, *Semiconductor Characterization System Technical Data, Keithley 4200-SCS.*, 2009. https://download.tek.com/datasheet/2199_4200-SCS_TechDataBook_RevP_053017.pdf.

[186] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, 1997.

[187] G. Dundar and K. Rose, “The effects of quantization on multilayer neural networks,” *IEEE Transactions on Neural Networks*, vol. 6, no. 6, pp. 1446–1451, 1995.

[188] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, “Going deeper in spiking neural networks: Vgg and residual architectures,” *Frontiers in Neuroscience*, vol. 13, p. 95, 2019.

[189] Y.-c. Wu and J.-w. Feng, “Development and application of artificial neural network,” *Wireless Personal Communications*, vol. 102, no. 2, pp. 1645–1656, 2018.

[190] M. Sivan, Y. Li, H. Veluri, Y. Zhao, B. Tang, X. Wang, E. Zamburg, J. Leong, J. Niu, U. Chand, and A. Thean, “All wse2 1t1r resistive ram cell for future monolithic 3d embedded memory integration,” *Nature Communications*, vol. 10, 11 2019.

[191] R. Berdan, A. Serb, A. Khiat, A. Regoutz, C. Papavassiliou, and T. Prodromakis, “A μ-controller-based system for interfacing selectorless rram crossbar arrays,” *IEEE Transactions on Electron Devices*, vol. 62, no. 7, pp. 2190–2196, 2015.

[192] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[193] J. Huang, S. Stathopoulos, A. Serb, and T. Prodromakis, “Neuropack: An algorithm-level python-based simulator for memristor-empowered neuro-inspired computing,” *Frontiers in Nanotechnology*, vol. 4, 2022.

[194] J. Huang, S. Stathopoulos, A. Serb, and T. Prodromakis, “A tool for emulating neuromorphic architectures with memristive models and devices,” 2022. arXiv:2207.07987.

[195] R. Gütig and H. Sompolinsky, “The tempotron: a neuron that learns spike timing–based decisions,” *Nature Neuroscience*, vol. 9, pp. 420–428, 2006.

[196] C. Frenkel, M. Lefebvre, and D. Bol, “Learning without feedback: Fixed random learning signals allow for feedforward training of deep neural networks,” *Frontiers in Neuroscience*, vol. 15, p. 20, 2021.

[197] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, “Neuronal dynamics: From single neurons to networks and models of cognition,” 2014.

[198] P. O'Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Frontiers in Neuroscience*, vol. 7, 2013.

[199] G. Hinton, "Coursera - neural networks for machine learning - geoffrey hinton," 2012.

[200] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013.

[201] S. Yin, S. K. Venkataramanaiah, G. K. Chen, R. Krishnamurthy, Y. Cao, C. Chakrabarti, and J.-s. Seo, "Algorithm and hardware design of discrete-time spiking neural networks based on back propagation with binary activations," in *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–5, 2017.

[202] Y. Messaris, S. Nikolaidis, A. Serb, S. Stathopoulos, I. Gupta, A. Khiat, and T. Prodromakis, "A tio2 reram parameter extraction method," pp. 1–4, 05 2017.

[203] S. Shin, K. Kim, and S.-M. Kang, "Compact models for memristors based on charge-flux constitutive relationships," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, pp. 590 – 598, 05 2010.

[204] C. Li, D. Belkin, Y. Li, P. Yan, M. Hu, N. Ge, H. Jiang, E. Montgomery, P. Lin, Z. Wang, W. Song, J. P. Strachan, M. Barnell, Q. Wu, R. S. Williams, J. J. Yang, and Q. Xia, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," 2018.

[205] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[206] L. Long and G. Fang, "A review of biologically plausible neuron models for spiking neural networks," *AIAA Infotech@ Aerospace 2010*, p. 3540, 2010.

[207] H. Cohen Duwek and E. Ezra Tsur, "Biologically plausible spiking neural networks for perceptual filling-in," in *Proceedings of the Annual Meeting of the Cognitive Science Society*, vol. 43, 2021.

[208] P. T. P. Tang, T.-H. Lin, and M. Davies, "Sparse coding by spiking neural networks: Convergence theory and computational results," 2017. arXiv:1705.05475.

[209] L. Cheng, Y. Liu, Z.-G. Hou, M. Tan, D. Du, and M. Fei, "A rapid spiking neural network approach with an application on hand gesture recognition," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 13, no. 1, pp. 151–161, 2021.

[210] S. Kim, S. Park, B. Na, and S. Yoon, "Spiking-yolo: Spiking neural network for energy-efficient object detection," in *AAAI*, 2020.

[211] W. W. Lee, S. L. Kukreja, and N. V. Thakor, "Cone: Convex-optimized-synaptic efficacies for temporally precise spike mapping," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 4, pp. 849–861, 2017.

[212] D. Reid, A. J. Hussain, and H. Tawfik, "Financial time series prediction using spiking neural networks," *PloS one*, vol. 9, no. 8, pp. e103656–e103656, 2014.

[213] C. P. N, K. Saboo, and B. Rajendran, "Composer classification based on temporal coding in adaptive spiking neural networks," in *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2015.

[214] M. J. Pearson, A. G. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche, "Implementing spiking neural networks for real-time signal-processing and control applications: A model-validated fpga approach," *IEEE Transactions on Neural Networks*, vol. 18, no. 5, pp. 1472–1487, 2007.

[215] S. Ghosh-Dastidar and H. Adeli, "A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection," *Neural Networks*, vol. 22, no. 10, pp. 1419–1431, 2009.

[216] E. Rueckert, D. Kappel, D. Tanneberg, D. Pecevski, and J. Peters, "Recurrent spiking networks solve planning tasks," *Scientific Reports*, vol. 6, no. 1, p. 21142, 2016.

[217] X. Wang, Z.-G. Hou, F. Lv, M. Tan, and Y. Wang, "Mobile robots' modular navigation controller using spiking neural networks," *Neurocomputing*, vol. 134, pp. 230–238, 2014. Special issue on the 2011 Sino-foreign-interchange Workshop on Intelligence Science and Intelligent Data Engineering (IScIDE 2011) Learning Algorithms and Applications.

[218] Z. Bing, C. Meschede, F. Röhrbein, K. Huang, and A. C. Knoll, "A survey of robotics control based on learning-inspired spiking neural networks," *Frontiers in Neurorobotics*, vol. 12, 2018.

[219] Y. Wang, Y. Zeng, J. Tang, and B. Xu, "Biological neuron coding inspired binary word embeddings," *Cognitive Computation*, vol. 11, no. 5, pp. 676–684, 2019.

[220] M. Białas, M. M. Mirończuk, and J. Mańdziuk, "Biologically plausible learning of text representation with spiking neural networks," in *Parallel Problem Solving from Nature – PPSN XVI*, (Cham), pp. 433–447, Springer International Publishing, 2020.

[221] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks," 2019. arXiv: 1901.09948.

[222] J. Huang, A. Serb, S. Stathopoulos, and T. Prodromakis, "Text classification in memristor-based spiking neural networks," 2022. arXiv:2207.13729.

[223] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2015.

[224] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2018. arXiv:1803.08375.

[225] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in Neuroscience*, vol. 11, 2017.

[226] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2015.

[227] D. Zambrano and S. M. Bohte, "Fast and efficient asynchronous neural computation with adapting spiking neural networks," 2016. arXiv:1609.02053.

[228] A. Lydia and S. Francis, "Adagrad - an optimizer for stochastic gradient descent," vol. Volume 6, pp. 566–568, 05 2019.

[229] B. Trevett, "pytorch-sentiment-analysis," 2021. https://github.com/bentrevett/pytorch-sentiment-analysis.