# Towards Multiplier-less Implementation of Neural Networks

**Amir Ardakani**

Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Doctor of Philosophy

*To my beloved family*

# Abstract

Artificial Intelligence (AI) has become profoundly embedded in contemporary life, with its applications proliferating across a wide array of domains. Central to AI are neural networks, which have markedly enhanced the capabilities of AI in areas such as computer vision and natural language processing. As neural networks scale in both size and computational complexity, the intelligent devices tasked with executing these networks face growing demands for computational and energy resources to ensure efficient and reliable performance. Consequently, resource-limited embedded devices, such as smartphones, encounter significant challenges in deploying state-of-the-art AI models. These devices frequently resort to cloud-based platforms, which necessitate continuous internet connectivity. However, cloud-based solutions pose several critical issues, including concerns over security, privacy, latency, and notably, their substantial environmental impact due to high energy consumption. This dissertation seeks to address these challenges by reducing the computational complexity of neural networks to facilitate their deployment on embedded devices. Specifically, it targets the primary source of computational burden and a major contributor to energy consumption in neural networks: high-precision multipliers (e.g., 16-bit or 8-bit multipliers).

We propose novel implementations of neural networks that either markedly reduce the bit-width of multipliers (to 4 bits or fewer) or entirely replace them with simpler logic operations (e.g., XNOR and shift operations). Moreover, reducing the bit-width of neural networks leads to decreased memory storage demands and reduced memory access, thereby contributing to a further reduction in overall energy consumption. In our initial implementation of neural networks, we present a novel approach for training multi-layer networks utilizing Finite State Machines (FSMs). In this approach, each FSM is interconnected with every FSM in both the preceding and subsequent layers. We demonstrate that the FSM-based network can effectively synthesize complex multi-input functions, such as 2D Gabor filters, and perform non-sequential tasks, such as image classification on stochastic streams, without the need for multiplications, given that FSMs are implemented solely through look-up tables. Building on

the FSMs' capability to handle binary streams, we propose an FSM-based model specifically designed for handling time series data, applicable to temporal tasks such as character-level language modeling. In our second implementation, we introduce an advanced stochastic computing (SC) representation termed the dynamic sign-magnitude (DSM) stream. This representation is specifically designed to enhance the precision of short-sequence SC-based multiplication. The DSM framework facilitates the substitution of conventional neural network multiplications with more efficient bitwise XNOR operations. By employing DSM, we achieve a substantial reduction in the required sequence length for SC-based neural networks, while maintaining accuracy levels comparable to existing methodologies. In our third implementation, we propose a new training framework for base-2 logarithmic quantization of neural networks. This framework quantizes weights into discrete power-of-two values by leveraging information about the network's weight distribution, specifically the standard deviation. This method allows us to replace computationally intensive high-precision multipliers with more efficient shift-add operations. Consequently, our quantized networks use approximately one-eighth the number of parameters compared to conventional high-precision networks, without compromising classification accuracy. Finally, in our latest implementation, we introduce a novel training framework that utilizes quantization techniques to facilitate the conversion between quantized networks and spiking neural networks (SNNs). SNNs are inherently devoid of multiplications, relying instead on addition and subtraction. This new framework offers an alternative approach for training SNNs. Specifically, we modify the SNN algorithm and mathematically demonstrate that after T time steps, the modified SNN approximates the behavior of a quantized network with T quantization intervals. This allows for the straightforward replacement of any SNN with its corresponding quantized network for training purposes. Given that the SNN and the quantized network share identical parameters, we can seamlessly transfer the parameters from the trained quantized network to the SNN without additional steps.

# Abrégé

L'intelligence artificielle (IA) est aujourd'hui profondément intégrée dans la vie contemporaine, avec des applications qui se multiplient dans un large éventail de domaines. Au cœur de l'IA se trouvent les réseaux de neurones, qui ont considérablement renforcé les capacités des systèmes intelligents, notamment dans les domaines de la vision par ordinateur et du traitement du langage naturel. Cependant, à mesure que les réseaux de neurones gagnent en taille et en complexité computationnelle, les dispositifs intelligents chargés de leur exécution doivent faire face à des exigences croissantes en matière de ressources de calcul et d'énergie pour garantir des performances efficaces et fiables. Ainsi, les dispositifs embarqués à ressources limitées, tels que les smartphones, rencontrent des difficultés majeures pour déployer des modèles d'IA de pointe. Ces dispositifs ont souvent recours à des plateformes cloud, qui nécessitent une connectivité internet constante. Toutefois, les solutions basées sur le cloud soulèvent plusieurs problématiques critiques, notamment en matière de sécurité, de confidentialité, de latence, ainsi que leur impact environnemental important, dû à une consommation énergétique élevée. Cette thèse vise à répondre à ces enjeux en réduisant la complexité computationnelle des réseaux de neurones afin d'en faciliter le déploiement sur des dispositifs embarqués. Plus précisément, elle s'attaque à la principale source de charge computationnelle — et à un facteur majeur de consommation d'énergie — dans les réseaux de neurones : les multiplicateurs de haute précision (par exemple, les multiplicateurs 16 bits ou 8 bits). Nous proposons de nouvelles implémentations de réseaux de neurones qui réduisent de manière significative la largeur de bits des multiplicateurs (jusqu'à 4 bits ou moins), ou les remplacent entièrement par des opérations logiques plus simples (par exemple, les opérations XNOR ou les décalages binaires). De plus, la réduction de la précision des réseaux de neurones permet de diminuer les besoins en mémoire de stockage ainsi que le nombre d'accès à la mémoire, contribuant ainsi à une baisse supplémentaire de la consommation énergétique globale. Dans notre première implémentation, nous présentons une nouvelle approche d'entraînement de réseaux multi-couches utilisant des machines à états finis (Finite State Machines, FSM). Dans cette architecture, chaque FSM est interconnectée avec toutes les FSM des couches précédentes et

suivantes. Nous démontrons que ce réseau basé sur des FSM peut synthétiser efficacement des fonctions complexes à entrées multiples, telles que les filtres de Gabor 2D, et exécuter des tâches non séquentielles, telles que la classification d'images à partir de flux stochastiques, sans recourir à des multiplications, les FSM étant implémentées uniquement à l'aide de tables de correspondance. En s'appuyant sur la capacité des FSM à traiter des flux binaires, nous proposons également un modèle basé sur les FSM spécifiquement conçu pour traiter des données temporelles, applicable à des tâches telles que la modélisation de langage au niveau des caractères. Dans notre seconde implémentation, nous introduisons une représentation avancée pour le calcul stochastique (Stochastic Computing, SC), que nous appelons flux dynamique en signe et magnitude (Dynamic Sign-Magnitude, DSM). Cette représentation a été spécialement conçue pour améliorer la précision des multiplications SC utilisant de courtes séquences. Le cadre DSM permet de remplacer les multiplications classiques des réseaux de neurones par des opérations XNOR binaires, beaucoup plus efficaces. Grâce à cette méthode, nous obtenons une réduction significative de la longueur de séquence nécessaire dans les réseaux SC, tout en maintenant un niveau de précision comparable à celui des approches existantes. Dans notre troisième implémentation, nous proposons un nouveau cadre d'entraînement pour la quantification logarithmique en base 2 des réseaux de neurones. Cette méthode quantifie les poids en valeurs discrètes correspondant à des puissances de deux, en exploitant l'information statistique de la distribution des poids, notamment leur écart type. Ce cadre permet de remplacer les multiplicateurs haute précision, coûteux en ressources, par des opérations de décalage et d'addition. Nos réseaux quantifiés utilisent ainsi environ huit fois moins de paramètres que les modèles haute précision classiques, tout en conservant une précision de classification équivalente. Enfin, dans notre implémentation la plus récente, nous introduisons un nouveau cadre d'entraînement utilisant des techniques de quantification pour faciliter la conversion entre réseaux quantifiés et réseaux de neurones impulsionnels (Spiking Neural Networks, SNN). Ces derniers sont naturellement dépourvus de multiplications, se reposant uniquement sur des additions et soustractions. Le cadre proposé offre une approche alternative pour l'entraînement des SNN. Plus précisément, nous modifions l'algorithme SNN et démontrons mathématiquement qu'après T pas de temps, le SNN modifié approxime le comportement d'un réseau quantifié avec T intervalles de quantification. Cela permet de remplacer aisément un SNN par son réseau quantifié équivalent pour l'entraînement. Étant donné que le SNN et le réseau quantifié partagent les mêmes paramètres, il est alors possible de transférer directement les paramètres du réseau quantifié entraîné vers le SNN, sans étape supplémentaire.

# Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Professor Warren J. Gross, for his unwavering support, mentorship, and encouragement throughout this journey. His profound knowledge, thoughtful guidance, and patience have been invaluable in helping me overcome challenges and ensuring the success of this work. I am deeply appreciative of the time and effort he has dedicated to my academic and professional development, and I consider myself truly fortunate to have had the opportunity to work under his supervision.

I would also like to extend my sincere gratitude to the members of my supervisory committee, Professor Brett H. Meyer and Professor James J. Clark, for their invaluable guidance and constructive feedback throughout this journey. In particular, I am especially grateful to Professor Brett H. Meyer for his steadfast emotional support and encouragement, which motivated me during challenging times and helped me persevere.

I wish to express my deepest appreciation to my brother, Arash Ardakani, whose support has been invaluable both personally and professionally throughout my Ph.D. journey. As both a labmate and a family member, he provided unwavering encouragement during the initial and most challenging phases of my studies. His insightful contributions were instrumental in the development of Chapter 2 of this dissertation. His guidance, collaboration, and belief in my abilities have been a cornerstone of my success, and I am profoundly thankful for his presence throughout this journey.

I would like to extend my heartfelt thanks to my labmates, Furkan Ercan, Thibaud Tonnellier, Nghia Doan, Syed Mohsin Abbas, Harsh Aurora, Adam Cavatassi, Jerry Ji, Mohammadreza Tayaranian Hosseini, Mohamed Abdelgawad, Charles Le, Hang Zhang, Marwan Jalaleddine, Jiajie Li, Ryan Seah, and Elie Mambou, for their camaraderie and insightful discussions throughout my research journey. Special thanks go to Mohammadreza Tayaranian Hosseini for his willingness to share knowledge, exchange ideas, and provide support. Your contributions have made my time in the lab both productive and enjoyable. I am truly grateful for the collaborative and friendly atmosphere we created together, which

has been a constant source of inspiration and motivation during my studies.

Finally, I would like to take this opportunity to express my deepest gratitude to my parents for their boundless love, unwavering support, and encouragement throughout my entire journey. They have always been my pillars of strength, instilling in me the confidence and perseverance to face challenges with resilience. Their sacrifices, guidance, and belief in my abilities have been a constant source of motivation, and I am forever indebted to them for their unconditional support.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**AI** Artificial Intelligence
**ANN** Artificial Neural Networks
**ASIC** Application-Specific Integrated Circuit
**BPC** Bits Per Character
**BN** Batch Normalization
**BPTT** Backpropagation Through Time
**CE** Cross-Entropy
**CLLM** Character-Level Language Modeling
**CNN** Convolutional Neural Networks
**DNNs** Deep Neural Networks
**DSM** Dynamic Signed-Magnitude
**FP** Full Precision
**FCNNs** Fully Connected Neural Networks
**FSM** Finite-State Machine
**FPGA** Field-Programmable Gate Array
**GD** Gradient Descent
**GRUs** Gated Recurrent Units
**GPU** Graphics Processing Units
**LFSRs** Linear-Feedback Shift Registers
**LIF** Leaky Integrate-and-Fire
**LSTMs** Long Short-Term Memory
**LK** Linux Kernel
**LUTs** Look-Up Tables
**MSE** Mean Squared Error
**NAS** Neural Architecture Search
**NN** Neural Network

**PT** Penn Treebank

**RNN** Recurrent Neural Network

**RBMs** Restricted Boltzmann Machines

**SC** Stochastic Computing

**SM** Sign-Magnitude

**SNG** Stochastic Number Generator

**SNN** Spiking Neural Network

**SOTA** State-of-the-Art

**WLFSM** Weighted Linear Finite-State Machine

**WP** War & Peace

# 1

# Introduction

The integration of artificial intelligence (AI) into our personal life has been expanding significantly, transforming how individuals interact with technology and manage everyday activities. AI-driven virtual assistants facilitate tasks such as scheduling, information retrieval, and controlling Internet of Things (IoT) devices, thereby enhancing efficiency and convenience. Personalized recommendation algorithms employed by streaming platforms, e-commerce websites, and social media platforms curate content and products based on user preferences, optimizing user experience and engagement. Furthermore, AI-powered health and fitness applications monitor biometric data, suggest exercise regimens, and promote healthier lifestyles through tailored recommendations. Language translation tools equipped with AI capabilities have also made cross-linguistic communication more accessible and effective. The adoption of state-of-the-art AI models on edge devices presents significant challenges, despite the potential benefits of reduced latency, enhanced privacy, and localized processing. One of the primary obstacles is the resource-constrained nature of edge devices, such as smartphones, IoT sensors, and embedded systems. These devices often lack the computational power, memory, and energy resources required to execute large-scale AI models, which are typically optimized for high-performance servers with abundant resources. This limitation necessitates model compression techniques, such as pruning [2], quantization [3], knowledge distillation [4],

neural architecture search (NAS) [5], hand-tuned tiny models [6–9], and multiplier-less designs [10–12], which can compromise model accuracy and performance. As the demand for AI-driven applications on edge devices grows, the development of efficient compression techniques tailored to these platforms is critical to enabling real-time, high-performance AI applications while ensuring energy efficiency and scalability.

Quantization is a technique employed to reduce the computational and memory requirements of deep learning models by representing weights and activations with lower-precision numerical formats, such as 8-bit integers, instead of the standard 32-bit floating-point representation. Neural network pruning is an optimization technique designed to reduce the computational complexity and memory footprint of deep learning models by removing redundant or less significant parameters, such as weights or neurons. Pruning methods are typically categorized into structured pruning, which removes entire components like channels or layers, and unstructured pruning, which eliminates individual weights based on certain criteria, such as magnitude or gradient sensitivity [13]. Knowledge distillation is a model compression technique wherein a smaller, simpler model (referred to as the "student") is trained to replicate the behavior of a more complex model (referred to as the "teacher"). By transferring the knowledge embedded in the teacher's outputs—often through soft probability distributions over class predictions or intermediate feature representations—the student model can achieve comparable performance while significantly reducing computational complexity and memory requirements. Knowledge distillation not only improves the efficiency of the student model but also enhances its generalization ability by leveraging the richer information encoded in the teacher's soft labels [14]. NAS is an automated approach for designing neural network architectures that aims to optimize performance metrics such as accuracy, efficiency, and scalability [5]. By leveraging search algorithms, NAS explores a predefined search space of possible architectures to identify optimal designs for specific tasks, eliminating the need for manual trial-and-error processes traditionally performed by human experts. Given the limited computational power, memory, and energy resources of edge devices, NAS for these platforms must prioritize architectural efficiency while maintaining high accuracy and minimizing latency. The search space for such architectures is typically constrained by factors such as model size, number of parameters, and computational complexity, with a focus on lightweight models that can operate within the power and memory limitations of edge devices. Hand-tuned tiny model design refers to the development of neural network architectures that are manually designed for efficiency, specifically targeting constraints in computational resources, memory, and power consumption, required for deployment on edge devices and

mobile platforms. These networks are designed to achieve high performance with reduced model size and computational complexity, addressing the challenges of resource-constrained environments. Techniques such as pruning, quantization, and knowledge distillation are commonly employed to reduce the number of parameters and operations required by the network, while maintaining or even enhancing accuracy [6–9].

Multiplierless neural network design is a specialized approach that focuses on reducing or entirely eliminating the use of multiplications within the network's computation, thereby optimizing the network for efficient hardware implementation, particularly in resource-constrained environments. Traditional neural networks rely heavily on multiplication operations, which can be computationally expensive and power-intensive, especially in hardware with limited processing and energy capabilities. By employing alternative operations, such as bitwise operations, addition, and shifts, multiplierless designs significantly reduce both the energy consumption and the hardware complexity required for neural network inference. For instance, techniques such as binary or ternary quantization [15], where weights and activations are constrained to binary or ternary values, allow the replacement of multiplications with simple logical operations.

## 1.1 Motivation

This dissertation aims to address the challenges of computational complexity in neural networks by developing efficient multiplierless architectures. Specifically, this research focuses on three key approaches: stochastic computing, shift-based multiplications, and spiking neural networks. These methodologies are designed to eliminate the reliance on power-intensive and computationally demanding binary-radix multiplications (e.g., 32-bit floating-point operations) during inference, thereby enabling more resource-efficient implementations suitable for resource-constrained environments.

Stochastic computing (SC) is a computational paradigm that represents data and performs operations using stochastic bit streams rather than conventional binary encoding [16]. In this approach, numerical values are encoded as the probability of a bit being "1" in a sequence, and arithmetic operations, such as addition, multiplication, and scaling, are executed using simple logic gates. This methodology offers significant advantages in terms of hardware simplicity [17], fault tolerance [18], and energy efficiency [19], making it particularly appealing for applications in low-power and resource-constrained environments, such as edge computing and neuromorphic systems. However, the precision of stochastic computing is inherently limited by the length of the bit streams, requiring trade-offs between accuracy and

computational latency [19].

Shift-based neural network design is an emerging approach aimed at enhancing the efficiency of neural networks by replacing computationally expensive multiplication operations with bit-shift operations, which are simpler and less resource-intensive [11, 20]. This design paradigm leverages the inherent properties of shift operations to perform scalar multiplications, thereby reducing power consumption and computational overhead. Shift-based architectures are particularly well-suited for deployment on resource-constrained devices, such as embedded systems and edge platforms, where energy efficiency and low latency are critical. Despite these advantages, challenges remain in maintaining model accuracy and generalization capabilities, as the replacement of multiplications with shifts can introduce precision loss and limit representational flexibility. For example, the recent state-of-the-art method in terms of classification accuracy performance, Sign-Sparse-Shift [11], requires the use of four sets of trainable parameters during the training phase to mitigate precision loss. However, this approach significantly increases memory requirements, posing challenges when training large-scale neural networks.

Spiking Neural Networks (SNNs) represent a biologically inspired computational paradigm that mimics the spike-based information processing of biological neurons. Unlike traditional artificial neural networks (ANNs), SNNs process and transmit information using discrete spikes or events, enabling asynchronous and event-driven computation. This approach offers significant advantages in terms of energy efficiency and temporal information processing, making SNNs particularly well-suited for applications in neuromorphic hardware [21]. On edge devices, SNNs are particularly suitable for tasks such as real-time sensory processing, anomaly detection, and autonomous systems, where low latency and energy efficiency are critical [22,23]. Despite their potential, SNNs face challenges, including the difficulty of training due to their non-differentiable spiking behavior and the need for specialized training algorithms such as surrogate gradient methods [24–26]. ANN-to-SNN conversion is a technique designed to leverage the well-established training methods of traditional ANNs while benefiting from the energy-efficient, event-driven nature of SNNs [27, 28]. In this approach, an ANN is first trained using standard backpropagation techniques with continuous activations, after which the trained model is converted into an equivalent SNN by mapping the activations and weights of the ANN to spiking neurons and synapses. This process typically involves replacing activation functions, such as ReLU, with spiking neuron models and calibrating parameters to maintain the accuracy of the original ANN during inference [29, 30]. While ANN-to-SNN conversion simplifies training, it introduces challenges such as accuracy degradation due

to temporal dynamics in SNNs and the need for precise normalization of input data and weights to ensure proper spike-rate representation [31]. Research in this area focuses on minimizing conversion-induced errors through techniques such as layer-wise optimization [30] and threshold tuning [32].

## 1.2 Objective

This dissertation seeks to address the main challenges associated with multiplierless designs: More specifically:

- Long sequence length required for maintaining classification accuracy in SC-based neural networks,
- Training challenges such as overparametrization and precision loss in shift-based neural networks, and
- Accuracy degradation induced by ANN-to-SNN conversion.

Specifically, the following topics are covered:

- Design of a multiplierless Weighted Linear Finite-State Machine-based neural network.
- Design of the Dynamic Sign-Magnitude Stochastic Stream for high-accuracy and low-latency stochastic computing based neural networks.
- Design of the Standard Deviation-based Quantization Framework for efficient training of high-accuracy Shift-based neural networks.
- Design of an ANN-to-SNN conversion framework to reduce the conversion error and latency.

## 1.3 Contribution and Thesis Outline

The contributions of this thesis can be summarized as follows:

**Chapter 2: Weighted Linear Finite-State Machine**

A finite-state machine (FSM) is a computation model to process binary strings in sequential circuits. Hence, a single-input linear FSM is conventionally used to implement complex single-input functions , such as tanh and exponentiation functions, in SC domain where continuous values are represented by sequences of random bits. In this Chapter, we introduce

a method that can train a multi-layer FSM-based network where FSMs are connected to every FSM in the previous and the next layer. We show that the proposed FSM-based network can synthesize multi-input complex functions such as 2D Gabor filters and can perform non-sequential tasks such as image classifications on stochastic streams with no multiplication since FSMs are implemented by look-up tables only. Inspired by the capability of FSMs in processing binary streams, we then propose an FSM-based model that can process time series data when performing temporal tasks such as character-level language modeling. Unlike long short-term memories (LSTMs) that unroll the network for each input time step and perform back-propagation on the unrolled network, our FSM-based model requires to backpropagate gradients only for the current input time step while it is still capable of learning long-term dependencies. Therefore, our FSM-based model can learn extremely long-term dependencies as it requires $1/l$ memory storage during training compared to LSTMs, where $l$ is the number of time steps. Moreover, our FSM-based model reduces the power consumption of training on a GPU by 33% compared to an LSTM model of the same size.

The content of this chapter has been presented in Advances in Neural Information Processing Systems (NeurIPS) conference [33], titled "Training linear finite-state machines" by Arash Ardakani, Amir Ardakani and Warren J. Gross. Dr. Arash Ardakani and I have contributed equally to the development of this work.

### Chapter 3: Dynamic Sign-Magnitude Stochastic Representation for Stochastic Computing

In this chapter, we propose a novel implementation of SC-based neural networks that utilizes two distinct types of stochastic streams. In this design, activations are represented using bipolar streams, while weights are encoded with sign-magnitude streams. To facilitate the multiplication of weights and activations from different formats, we introduce a new type of stochastic stream, referred to as dynamic signed-magnitude (DSM). This approach enables both the weights and activations to share the same Stochastic Number Generator (SNG), thereby reducing the number of required SNGs and minimizing the overall hardware footprint. Experimental results on the MNIST dataset demonstrate that our SC-based neural networks outperform existing SC-based models in terms of processing latency (i.e., sequence length), achieving reductions of up to 16×, while maintaining accuracy levels comparable to existing methodologies.

The content of this chapter has been published in IEEE Design & Test journal [1], titled "Training binarized neural networks using ternary multipliers", by Amir Ardakani, Arash

Ardakani and Warren Gross.

**Disclaimer:**The content of this chapter is adapted and extended from our prior work published in the IEEE Design & Test journal ©IEEE 2021 [1]. The text has been rephrased and rewritten to better articulate our contributions. Additionally, the work has been expanded to include a detailed mathematical explanation, accompanied by an example, to illustrate the proposed dynamic sign-magnitude stochastic representation.

**Chapter 4: Shift-Net Neural Network**

In this chapter, we present a new quantization framework that utilizes the standard deviation of network weight and activation distributions. We also introduce an improved training strategy for base-2 logarithmic quantization, which maps weights to discrete power-of-two values. Base-2 logarithmic quantization replaces high-precision multipliers with efficient shift-and-add operations, significantly reducing computational overhead. Our training method lowers memory usage and shortens training time (in terms of epochs). Experimental results on CIFAR-10 and ImageNet show that our method outperforms existing techniques, achieving higher accuracy with 3-bit weights and activations compared to full-precision models.

Part of this chapter has been presented in Edge Intelligence Workshop (EIW) [34], titled "Standard Deviation-Based Quantization for Deep Neural Networks", by Amir Ardakani, Arash Ardakani, Brett Meyer, James J. Clark and Warren J. Gross. The extended version of this work will be submitted to a journal in near future.

**Chapter 5: Towards Lossless ANN-to-SNN Conversion**

In this chapter, we demonstrate that the behavior of neurons in SNNs closely resembles the activation patterns observed in quantized neural networks. Specifically, we establish that the activation function of an SNN corresponds to the activation function of a neural network quantized using the floor function. Drawing on this insight, we propose a conversion framework for SNNs, wherein a quantized neural network serves as the source for the conversion process. Within this framework, the quantization method introduced in Chapter 4 is employed. We evaluate the proposed method on the CIFAR-10, and ImageNet datasets, demonstrating that it surpasses the state-of-the-art ANN-to-SNN conversion techniques and directly trained SNNs in terms of both accuracy and inference time-steps. The content of this work will be submitted to a journal in near future.

**Chapter 6: Conclusions and Future Work**

This chapter addresses the conclusions of this thesis, and future research directions are suggested.

## 1.4 Contributions of the Authors

This dissertation presents original work in the field of stochastic computing-based, quantized and spiking neural networks by Amir Ardakani.

In Chapter 2, a method for training a multi-layer weighted finite-state machine (WFSM)-based network using backpropagation is introduced. Previously, Ardakani et al. proposed a training approach for WFSMs by minimizing the mean square error of a cost function to synthesize complex arithmetic computations on stochastic streams [35]. Following Dr. Arash Ardakani's suggestion, I developed an algorithm to train WFSMs in a multi-layer network using gradient descent and backpropagation. To assess the performance of the proposed method, Dr. Arash Ardakani and I collaboratively developed Python scripts and conducted benchmarking on both non-sequential and temporal tasks. The results of this research were presented at the Advances in Neural Information Processing Systems (NeurIPS) conference [33] in a paper titled *"Training Linear Finite-State Machines"*, authored by Arash Ardakani, Amir Ardakani, and Warren J. Gross. Dr. Arash Ardakani and I are co-first authors and contributed equally to the development of this work. In Chapter 2, I have included a revised version of the originally presented work with the permission of Dr. Arash Ardakani.

In Chapter 3, I introduce a novel type of stochastic sequence, termed dynamic signed-magnitude (DSM), designed to facilitate the multiplication of weights and activations from different stochastic stream formats. The content of this chapter has been published in the *IEEE Design & Test* journal [1], in a paper titled *"Training Binarized Neural Networks Using Ternary Multipliers"*, authored by Amir Ardakani, Arash Ardakani, and Warren J. Gross. Dr. Arash Ardakani and Professor Warren J. Gross provided valuable suggestions to enhance the experimental framework and contributed to the revision of the original published work. In Chapter 3, I have included a revised and extended version of this work. Additionally, the chapter expands on the original publication by providing a detailed mathematical explanation, accompanied by an illustrative example, to further clarify the proposed dynamic signed-magnitude stochastic representation.

In Chapter 4, I introduce a base-2 logarithmic quantization scheme that maps weights to discrete power-of-two values. The initial experimental results were presented at the Edge

Intelligence Workshop (EIW) [34] in a paper titled *"Standard Deviation-Based Quantization for Deep Neural Networks"*, authored by Amir Ardakani, Arash Ardakani, Brett Meyer, James J. Clark, and Warren J. Gross. Dr. Arash Ardakani, Professor Brett Meyer, Professor James J. Clark, and Professor Warren J. Gross contributed to the revision of the workshop paper. This chapter presents the original work developed for this dissertation.

In Chapter 5, I propose a new conversion framework for SNNs. This chapter presents the original work developed for this dissertation. Professor Warren J. Gross provided valuable suggestions for the revision of this chapter.

## 1.5 Publication

Here is a list of published work during my Ph.D. program at McGill University.

1. A. Ardakani, **A. Ardakani**, and W. Gross, "Training linear finite-state machines," Advances in Neural Information Processing Systems, vol. 33, pp. 7173–7183, 2020.

2. A. Ardakani, **A. Ardakani**, and W. J. Gross, "A regression-based method to synthesize complex arithmetic computations on stochastic streams," in 2020 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2020, pp. 1–5.

3. **A. Ardakani**, A. Ardakani, and W. J. Gross, "training binarized neural networks using ternary multipliers," IEEE Design & Test, vol. 38, no. 6, pp. 44–52, 2021

4. **A. Ardakani**, A. Ardakani, and W. J. Gross, "Fault-tolerance of binarized and stochastic computing-based neural networks," in 2021 IEEE Workshop on Signal Processing Systems (SiPS). IEEE, 2021, pp. 52–57.

5. **A. Ardakani**, A. Ardakani, B. Meyer, J. J. Clark, and W. J. Gross, "Standard deviation-based quantization for deep neural networks," in Edge Intelligence Workshop, 2022.

6. A. Ardakani, Z. Ji, **A. Ardakani**, and W. Gross, "The Synthesis of XNOR Recurrent Neural Networks with Stochastic Logic," in Thirty-third Conference on Neural Information Processing Systems, 2019.

7. S. M. Sadaghiani, **A. Ardakani**, and S. Bhadra, "Ambient light-driven wireless wearable finger patch for monitoring vital signs from ppg signal," IEEE Sensors Journal, 2023.

8. C. Le, A. Ardakani, **A. Ardakani**, H. Zhang, Y. Chen, J. Clark, B. Meyer, and W. Gross, "Efficient two-stage progressive quantization of BERT," in Proceedings of The Third Workshop on Simple and Efficient Natural Language Processing (SustaiNLP), 2022, pp.1–9.

# 2

# Training Linear Finite-State Machines

## 2.1 Introduction

This chapter introduces a method for training finite-state machines (FSMs). An FSM is a mathematical model of computation characterized by a finite set of states and transitions between these states. The primary function of an FSM is to sequentially transition between states in a systematic manner, executing predefined actions upon each state transition. Given that FSMs are inherently designed to process sequential data, we utilize stochastic computing (SC), which converts continuous values into bit streams, to enable FSMs to perform non-sequential tasks. The resulting system is referred to as an FSM-based network.

The FSM-based network consists exclusively of weighted linear finite-state machines (WLFSMs) and is constructed by stacking multiple layers, wherein each WLFSM is fully connected to every WLFSM in both the preceding and subsequent layers. In this architecture, each state of a WLFSM is associated with a weight, and outputs are generated by sampling the weight corresponding to the current state. Importantly, the FSM-based network is designed to operate entirely within the SC domain, performing inference computations on bit streams without the need for multiplication operations. To facilitate the training of FSM-based networks, we derive a function based on the steady-state conditions of linear

FSMs that computes the state occurrence probabilities for each FSM. We mathematically demonstrate the invertibility of this function, enabling the derivation of the derivative of the FSM's computational function with respect to its input. This property allows for the training of deep FSM-based networks. The proposed FSM-based network is then applied to two distinct tasks: the synthesis of multi-input complex functions and image classification. Unlike conventional methods, which rely on a single WLFSM to approximate single-input complex functions [36–38], we demonstrate that the FSM-based network is capable of approximating multi-input complex functions, such as 2D Gabor filters, using only linear FSMs. Furthermore, we evaluate the FSM-based network on the MNIST dataset [39] for classification tasks. The results show that the FSM-based network significantly outperforms SC-based counterparts of equivalent size in terms of accuracy, achieving superior performance while requiring only half the number of operations.

We further introduce an FSM-based model designed to perform temporal tasks. This model draws inspiration from sequential digital circuits, where FSMs serve as memory elements (e.g., registers) to store the model's state [40]. In addition to weighted linear finite-state machines (WLFSMs), the FSM-based model incorporates fully connected networks that function as a transition mechanism and an output decoder, analogous to combinational logic in sequential circuits. The transition function governs the transition between states, while the output function facilitates decision-making based on the model type—either using the present state alone, as in a *Moore machine*, or both the present state and current input, as in a *Mealy machine*. In this architecture, the next state is determined by the combination of the present state and the current input. Consequently, gradients are backpropagated for the current time step only. This approach stands in contrast to widely used recurrent neural network (RNN) variants, such as long short-term memory (LSTM) networks and gated recurrent units (GRUs) [41], which unroll the network across all time steps and backpropagate gradients through the entire unrolled sequence. As a result, the FSM-based model requires only $1/l$ memory elements to store intermediate values, where $l$ represents the number of time steps. This reduction leads to a 33% decrease in GPU power consumption during training compared to an LSTM model of equivalent size. Moreover, we demonstrate that the FSM-based model can effectively capture extremely long-range dependencies, such as those found in sequences with lengths up to 2500. This capability is validated through the character-level language modeling (CLLM) task, where the FSM-based model shows robust performance in handling extensive temporal dependencies.

## 2.2 Disclaimer

The content of this chapter is adapted from our prior work presented at the Advances in Neural Information Processing Systems (NeurIPS) conference [33]. The text has been restructured and refined to more clearly articulate our contributions. Previously, Ardakani et al. proposed a training approach for WFSMs by minimizing the mean square error of a cost function to synthesize complex arithmetic computations on stochastic streams [35]. Following Dr. Arash Ardakani's suggestion, I developed the algorithms to train WFSMs in a multi-layer network using gradient descent and backpropagation. To assess the performance of the proposed method, Dr. Arash Ardakani and I collaboratively developed Python scripts and conducted benchmarking on both non-sequential and temporal tasks. The results of this research were presented at the Advances in Neural Information Processing Systems (NeurIPS) conference [33] in a paper titled *"Training Linear Finite-State Machines"*, authored by Arash Ardakani, Amir Ardakani, and Warren J. Gross. Dr. Arash Ardakani and I are co-first authors and contributed equally to the development of this work. In this chapter, I have included a revised version of the originally presented work with the permission of Dr. Arash Ardakani.

## 2.3 Preliminaries

Here, we present a brief overview of the mathematical operations within the SC domain. For a comprehensive background on SC and its foundational principles, please refer to Appendix A. In the numerical system of SC, continuous values are represented as the frequency of ones in random bit streams [42]. This representation enables arithmetic operations to be performed using simple bit-wise operations directly on the bit streams. Due to the nature of SC, a single bit-flip in a stochastic stream results in only a marginal change in the continuous value represented by the stream, allowing SC-based implementations to tolerate small errors effectively. Consequently, SC-based systems offer ultra low-cost, fault-tolerant hardware solutions for a wide range of applications [43].

For a continuous value $x \in [0, 1]$, its corresponding stochastic stream $X = \{x_l, x_{l-1}, \ldots, x_2, x_1\}$ of length $l$ in SC's *unipolar* format is generated such that:

$$\mathbb{E}[x_t] = x, \tag{2.1}$$

where $\mathbb{E}[x_t]$ denotes the expected value of the random binary variable $x_t \in \{0, 1\}$. In SC's *bipolar* format, a continuous value $x \in [-1, 1]$ is encoded as a sequence of random binary

variables $x_t \in \{0, 1\}$ such that:

$$\mathbb{E}[x_t] = (x + 1)/2. \tag{2.2}$$

Throughout this chapter, bold uppercase letters (e.g., $\mathbf{X}$) will denote stochastic streams, while subscripted lowercase letters (e.g., $x_t$) will represent individual elements within a stochastic sequence.

Arithmetic operations in SC are implemented through bit-wise logical operations. For two independent stochastic streams $\mathbf{A}$ and $\mathbf{B}$, multiplication is performed using the bit-wise AND operation in the unipolar format and the bit-wise XNOR operation in the bipolar format [43]. Additions in SC are performed using scaled adders, which adjust the result of the addition to fit within the permissible ranges of $[0, 1]$ in the unipolar format and $[-1, 1]$ in the bipolar format. The scaled adder employs a multiplexer to compute the sum of two stochastic streams, $\mathbf{A}$ and $\mathbf{B}$. The multiplexer's output, $\mathbf{C}$, is given by:

$$c_t = a_t \cdot s_t + b_t \cdot (1 - s_t), \tag{2.3}$$

where " $\cdot$ " denotes the bit-wise AND operation, and $\mathbf{S}$ is a unipolar stochastic stream representing a continuous value. When $\mathbf{S}$ represents the value 0.5 (i.e., $\mathbb{E}(s_t) = 0.5$) the expected value of $c_t$ is equal to the average of the expected values of $a_t$ and $b_t$, expressed as:

$$\mathbb{E}(c_t) = \frac{\mathbb{E}[a_t] + \mathbb{E}[b_t]}{2}. \tag{2.4}$$

In SC, complex functions are traditionally implemented using linear FSMs [44]. A linear FSM consists of a finite set of states arranged in a sequential, linear structure. The general structure of a linear FSM with a set of $N$ states (i.e., $\{\psi_0, \psi_1, \ldots, \psi_{N-1}\}$) is illustrated in Figure 2.1(a). Conceptually, a linear FSM can be interpreted as a saturating counter that increments or decrements its state value without exceeding its maximum or minimum boundaries, respectively. State transitions within linear FSMs occur based on the current entry of the input stream. Specifically, if the current input entry ($x_t \in \{0, 1\}$ for $t \in \{0, 1, \ldots, l\}$) from the input stream $\mathbf{X} \in \{0, 1\}^l$ is 0, the state value decreases by one. Conversely, if the current input entry is 1, the state value increases by one.

## 2.4 Related Work

As an initial attempt to implement complex functions within the SC domain, Brown and Card introduced two FSM-based functions in [44]: the hyperbolic tangent (tanh) and exponentiation

(exp)functions. They demonstrated that if an $N$-state linear FSM outputs 1 for state values greater than or equal to $N/2$ and 0 otherwise, the expected value of the FSM's output approximates $\tanh(Nx/2)$ for an input stochastic stream $\mathbf{X}$ representing the continuous value $x$. Similarly, the function $\exp(-2Gx)$ can be approximated if the FSM outputs 1 for state values less than $N - G$ and 0 otherwise.

Li et al. later introduced weighted linear FSMs (WLFSMs), where each state is associated with a weight [38]. In WLFSMs, a binary output is generated by sampling from the weight corresponding to the current state, as illustrated in Figure 2.1(b). To implement a single-input function using a WLFSM, Li et al. formulated the task as a quadratic programming problem and employed numerical methods to determine the weights. Specifically, the quadratic programming problem was solved for all guiding stream values between 0 and 1 with a step size of 0.001, resulting in 1000 solutions. The solution with the minimum mean squared error (MSE) was selected. Li et al. further extended this work in [37] by introducing greater design flexibility to implement more complex functions. They proposed a two-dimensional FSM topology with $N \times M$ states, where $N$ and $M$ denote the number of states in each row and column, respectively. This topology enabled the implementation of more sophisticated functions.

Another approach to implementing complex functions involves approximating them with simpler functions that can be realized using linear FSMs. For example, Onizawa et al. in [45] approximated a sine function by combining several tanh functions. This approximation allows the sine function to be realized through multiple linear FSMs, each implementing a tanh function. Similarly, a Gabor filter function can be implemented by multiplying the approximated sine function with a Gaussian function, where the Gaussian function itself is realized using a linear FSM that approximates the exponentiation function in SC. More recently, Ardakani et al. proposed a regression-based approach to determine the weights of FSMs [36]. This method outperforms numerical synthesis techniques in terms of mean squared error (MSE), providing a more accurate and efficient solution for implementing complex functions.

## 2.5 FSM-Based Networks

In this section, we propose a method that enables the backpropagation of gradients in FSM-based networks. The FSM-based network is structured into layers of WLFSMs, where each WLFSM is fully connected to all WLFSMs in the preceding and succeeding layers, except for the first layer. In the first layer, each input is exclusively connected to a single WLFSM, as

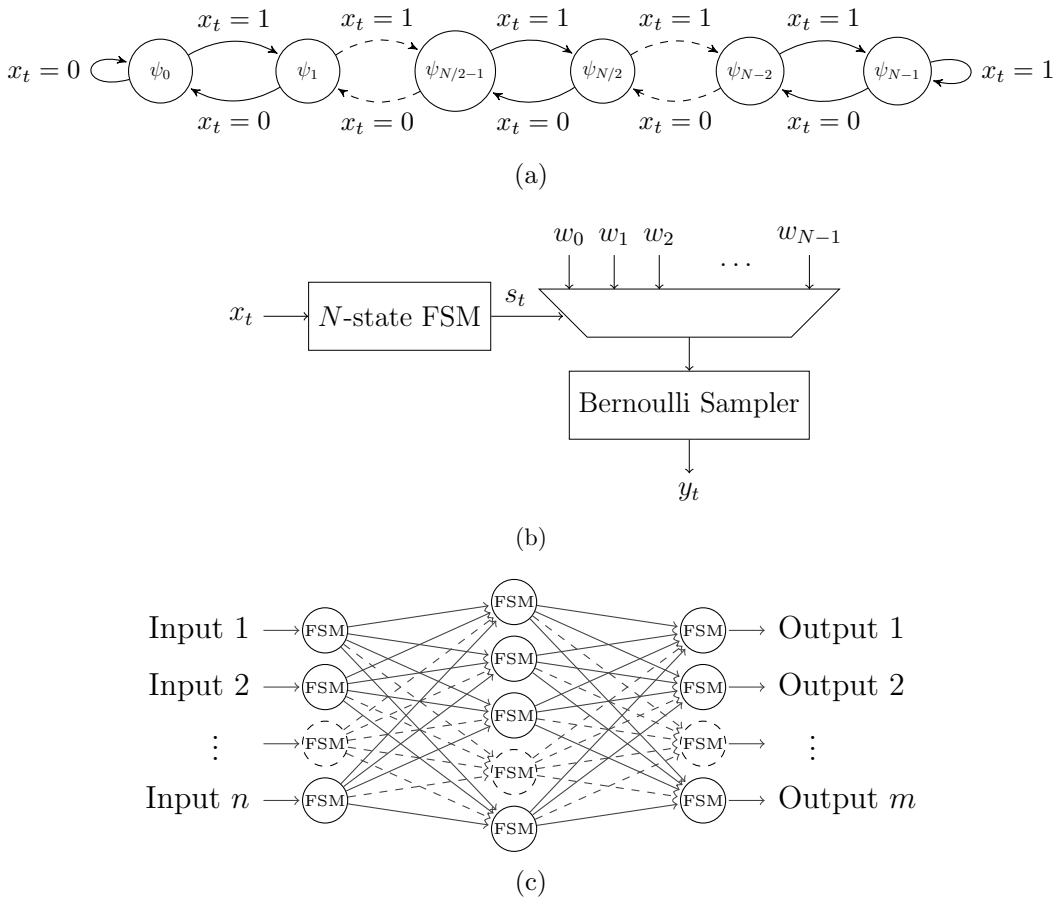



Figure 2.1: (a) A WLFSM with $N$ states where $x_t$ denotes the $t^{\text{th}}$ entry of the input stream $\mathbf{X} \in \{0,1\}^l$ for $t \in \{1,2,\ldots,l\}$. (b) An architecture implementing the WLFSM with $N$ states. (c) A general form of an FSM-based network.

depicted in Figure 2.1(c). For each WLFSM unit, the inputs are first aggregated and scaled to fit within the permissible range of the SC domain using a scaled adder. The output of the scaled adder is then passed to the WLFSM for further processing. In this configuration, the primary computations in FSM-based networks consist of additions and weight indexing operations. The weight indexing operations in WLFSMs are implemented using look-up tables (LUTs), which provide a significant advantage for hardware implementations on LUT-based platforms such as field-programmable gate arrays (FPGAs).

### 2.5.1 Backpropagation Method

To enable the backpropagation of gradients through FSM-based networks, we first define the forward computations of a WLFSM with $N$ states as follows:

$$y_t = \text{Bernoulli}\left(\frac{w_{s_t} + 1}{2}\right), \tag{2.5}$$

where $s_t \in \{0, 1, \ldots, N-1\}$ represents the state value, and $y_t \in \{0, 1\}$ denotes the output value corresponding to the input $x_t \in \{0, 1\}$ at time $t$, for $t \in \{1, 2, \ldots, l\}$. Bernoulli denotes the Bernoulli distribution applied to each element of $w_{s_t}$. The FSM is also associated with a set of weights, denoted as $\{w_0, w_1, \ldots, w_{N-1}\}$. By performing the computations for each individual input entry of the input stochastic stream $\mathbf{X} \in \{0, 1\}^l$, a stochastic output stream $\mathbf{Y} \in \{0, 1\}^l$ is generated, which represents the continuous value $y \in \mathbb{R}$ in bipolar format, such that $y = 2 \times \mathbb{E}(y_t) - 1$. However, training FSM-based networks on stochastic streams is $l$-times slower than conventional full-precision training methods. Consequently, we propose training FSM-based networks using the continuous values derived from the stochastic streams, while retaining stochastic bit streams for inference computations.

Given the occurrence probability (i.e., selection frequency) of the state $\psi_i$ as $p_{\psi_i}$ for $i \in \{0, 1, \ldots, N-1\}$, the continuous value of the WLFSM's output (denoted $y \in [-1, 1]$) can be expressed as:

$$y = \sum_{i=0}^{N-1} p_{\psi_i} \times w_{\psi_i}, \tag{2.6}$$

where the sum is taken over all states, and the probability $p_{\psi_i}$ corresponds to the occurrence probability of state $\psi_i$. This equation holds as the length of the stochastic streams tends to infinity ($l \to \infty$). In the steady state, the probability of the state transition from $\psi_{i-1}$ to $\psi_i$ must equal the probability of the transition from $\psi_i$ to $\psi_{i-1}$. This relationship is given by:

$$p_{\psi_i} \times (1 - p_x) = p_{\psi_{i-1}} \times p_x, \tag{2.7}$$

where $p_x$ is $(x + 1)/2$, with $x$ representing the continuous input value. In this context, the weight associated with $\psi_i$ is selected with probability $p_x$ during the forward state transition (i.e., from $\psi_{i-1}$ to $\psi_i$), while the weight associated with $\psi_{i-1}$ is selected with probability $1 - p_x$ during the backward state transition (i.e., from $\psi_i$ to $\psi_{i-1}$). Consequently, the derivative of the forward transition probability with respect to $x$ is 1, while the derivative of the backward

transition probability with respect to $x$ is $-1$ in the steady state. Thus, we have the relation:

$$\frac{\partial p_{\psi_i}}{\partial x} = -\frac{\partial p_{\psi_{i-1}}}{\partial x}. \tag{2.8}$$

Furthermore, the total occurrence probability of all states must sum to unity, i.e.,

$$\sum_{i=0}^{N-1} p_{\psi_i} = 1. \tag{2.9}$$

Using the relationships in Equation (2.7) and Equation (2.9), the general form of the occurrence probability is derived as:

$$p_{\psi_i} = \frac{\left(\frac{p_x}{1-p_x}\right)^i}{\sum_{j=0}^{N-1}\left(\frac{p_x}{1-p_x}\right)^j}. \tag{2.10}$$

Given Equation (2.6) and Equation (2.10), the weights of the WLFSM can be learned for the purpose of implementing a single-input complex function using linear regression. To extend the use of WLFSMs in a multi-layer network, it is necessary to compute the derivative of $p_{\psi_i}$ with respect to the input $x$. To achieve this, we first derive the inverse function for $p_{\psi_i}$, which will then be used to determine the derivative of $p_{\psi_i}$ with respect to the continuous value of the input stream $\mathbf{x}$. To compute the inverse function for $p_{\psi_i}$, we trained a single WLFSM to implement a linear function, where the outputs are equivalent to its inputs. We observed that the weights of the WLFSM with $N$ states alternate between $-1$ and 1, as expressed in:

$$x = \sum_{i=0}^{N-1}(-1)^{i+1} p_{\psi_i}, \tag{2.11}$$

where $i \in \{0, 1, \ldots, N-1\}$. To validate Equation (2.11), we utilize the geometric series sum formula [46], given by:

$$\sum_{i=0}^{N-1} r^i = \frac{1-r^N}{1-r}, \tag{2.12}$$

and the alternating series sum formula:

$$\sum_{i=0}^{N-1}(-1)^{i+1} r^i = \frac{(-1)^N r^N - 1}{1+r}, \tag{2.13}$$

where $r$ represents the common ratio. Using Equation (2.12) and Equation (2.13), we can rewrite the right-hand side of Equation (2.11) as:

$$\begin{aligned}
\sum_{i=0}^{N-1}(-1)^{i+1}p_{\psi_i} &= \sum_{i=0}^{N-1}\frac{(-1)^{i+1}\left(\frac{p_x}{1-p_x}\right)^i}{\sum_{j=0}^{N-1}\left(\frac{p_x}{1-p_x}\right)^j} \\
&= \frac{1}{\sum_{j=0}^{N-1}\left(\frac{p_x}{1-p_x}\right)^j}\sum_{i=0}^{N-1}(-1)^{i+1}\left(\frac{p_x}{1-p_x}\right)^i \\
&= \frac{1-\frac{p_x}{1-p_x}}{1-\left(\frac{p_x}{1-p_x}\right)^N} \times \frac{(-1)^N\left(\frac{p_x}{1-p_x}\right)^N - 1}{1+\frac{p_x}{1-p_x}} \\
&= \frac{1-(-1)^N\left(\frac{p_x}{1-p_x}\right)^N}{1-\left(\frac{p_x}{1-p_x}\right)^N} \times (2p_x-1) \overset{\text{for even}N}{=} (2p_x-1) = x. \qquad (2.14)
\end{aligned}$$

Thus, we have mathematically proven the validity of Equation (2.11), which was hypothesized based on the synthesis of a linear function using a WLFSM. Next, by differentiating Equation (2.9) and Equation (2.11) with respect to $x$, we obtain the following system of differential equations:

$$\frac{\partial p_{\psi_i}}{\partial x} = -\frac{\partial p_{\psi_{i-1}}}{\partial x}, \quad \sum_{i=0}^{N-1}\frac{\partial p_{\psi_i}}{\partial x} = 0, \quad \sum_{i=0}^{N-1}(-1)^{i+1}\frac{\partial p_{\psi_i}}{\partial x} = 1, \qquad (2.15)$$

where $i \in \{0, 1, \ldots, N-1\}$. Solving this system of equations yields the derivative of $p_{\psi_i}$ with respect to $x$:

$$\frac{\partial p_{\psi_i}}{\partial x} = \frac{(-1)^{i+1}}{N}, \qquad (2.16)$$

which can then be used to backpropagate gradients in FSM-based networks.

**Algorithm 1:** Pseudo code of the forward computations of training in FSM-based networks. $L$ is the number of layers including the output layer. The training loss is denoted as $\mathbb{C}$. $N$ denotes the number of states in FSMs. The Clamp function replaces the values greater than 1 and less than $-1$ with 1 and $-1$, respectively.

**Data:** An input minibatch of $\mathbf{x}^0 \in [-1, +1]^{d_b \times d_{x^0}}$, a target minibatch of $\overline{\mathbf{y}} \in [-1, +1]^{d_b \times d_{x^L}}$, the occurrence probability of the state $\psi_i$ as $\mathbf{p}^k_{\psi_i} \in [0, 1]^{d_b \times d_{x^k}}$, the occurrence probability of all the state as $\mathbf{p}^k_{\psi} \in [0, 1]^{d_b \times N d_{x^k}}$ and weights $\mathbf{W}^k \in [-1, +1]^{d_{x^k} \times d_{x^{k+1}}}$ for $k \in \{0, \ldots, L-1\}$ and $i \in \{0, \ldots, N-1\}$.

**for** $k = 0 : L - 1$ **do**
  **for** $i = 0 : N - 1$ **do**
    $$\mathbf{p}^k_{\psi_i} = \frac{\left(\frac{1+\mathbf{x}^k}{1-\mathbf{x}^k}\right)^i}{\sum_{j=0}^{N-1}\left(\frac{1+\mathbf{x}^k}{1-\mathbf{x}^k}\right)^j}$$
  **end**
  $\mathbf{p}^k_{\psi} = [\mathbf{p}^k_{\psi_0}, \mathbf{p}^k_{\psi_1}, \ldots, \mathbf{p}^k_{\psi_{N-1}}]$
  $$\mathbf{x}^{k+1} = \frac{\left(\mathbf{p}^k_{\psi}\text{Clamp}(\mathbf{W}^k, -1, 1)\right)}{d_{x^k}}$$
**end**
Compute loss $\mathbb{C}$ given $\mathbf{x}^L$ and $\overline{\mathbf{y}}$

### 2.5.2 Training Details

This section describes the training and inference procedures for FSM-based networks. During the training phase, the computations are performed in single-precision floating-point format, whereas the inference phase is executed using stochastic bit streams. The details of the forward computations during training are provided in Algorithm 1. In these forward computations, the occurrence probability of each state is calculated using Equation (2.10). While it is also possible to compute the occurrence probability by performing forward computations directly on stochastic bit streams, this approach is computationally expensive and significantly increases the training time. To ensure that the weights remain within the bipolar range of SC during inference, we constrain the weights to lie between $-1$ and 1 during the forward propagation in training.

The backward computations involved in training are outlined in Algorithm 2. In these computations, the gradients for the FSM-based layers are backpropagated using Equation (2.16). The choice of loss function $\mathbb{C}$ is dependent on the specific task being targeted by the FSM-

based network. It is important to note that the number of states $N$ in the FSM-based networks must be an even natural number, as described in Equation (2.14).

In contrast to the training phase, the inference computations are carried out on stochastic bit streams. The details of the inference procedure are provided in Algorithm 3. Since the output vector $\mathbf{o}_t^k$ of the FSMs is one-hot encoded, its multiplication with a binary weight sample $\mathbf{W}^k$ essentially involves indexing operations. Specifically, the main operations during inference consist of indexing and addition, making FSM-based networks multiplication-free. Furthermore, no separate nonlinear activation function is necessary when using FSM-based networks. FSMs themselves can be considered as nonlinear activation functions, capable of approximating the required non-linearity during the training process. It is noteworthy that FSMs are commonly employed to approximate nonlinear functions, such as the hyperbolic tangent (tanh) and exponential functions (exp), in the SC domain (see Section 2.3). These characteristics make FSM-based networks particularly well-suited for applications where ultra-low-cost implementations of inference computations are required.

### 2.5.3 Applications of FSM-Based Networks

#### 2D Gabor Filter

As an initial application of FSM-based networks, we demonstrate their ability to synthesize 2D Gabor filters. During the training phase, the forward computations are carried out using Equation (2.10), while the backward computations are performed using Equation (2.16). In contrast, the inference phase leverages stochastic bit streams for computation.

The imaginary part of a 2D Gabor filter is defined as:

$$g_{\sigma,\gamma,\theta,\omega}(x,y) = \exp\left(-\frac{\overline{x}^2 + \gamma^2\overline{y}^2}{2\sigma^2}\right)\sin(2\omega\overline{x}), \tag{2.17}$$

where $\overline{x} = x\cos\theta + y\sin\theta$ and $\overline{y} = -x\sin\theta + y\cos\theta$. The parameters $\sigma$, $\gamma$, $\theta$ and $\omega$ represent the standard deviation of the Gaussian envelope, the spatial aspect ratio, the orientation of the normal to the parallel stripes of the Gabor filter, and the spatial angular frequency of the sinusoidal component, respectively. Figure 2.2 presents the simulation results of FSM-based networks implementing a set of 2D Gabor filters used in the HMAX model [47]. To generate these results, we trained a three-layer FSM-based network of size 4 (with the network configuration $2-4-4-1$), where each WLFSM contained four states (i.e., $N = 4$). The network thus comprised 10 WLFSMs, each with 4 states, and a total of 112 weights. The Mean Squared Error (MSE) was used as the loss function, and Adam was utilized as the

**Algorithm 2:** Pseudo code of the backward computations of training in FSM-based networks. $L$ is the number of layers including output layer. The training loss is denoted as $\mathbb{C}$. $N$ and $\eta$ denote the number of states in FSMs and the learning rate, respectively. The gradient of parameters w.r.t. $\mathbb{C}$ is denoted by " $\hat{}$ " over their corresponding symbols.

**Data:** Gradients of activations as $\hat{\mathbf{x}}^k \in \mathbb{R}^{d_b \times d_{x^k}}$, the occurrence probability of the state $\psi_i$ as $\hat{\mathbf{p}}^k_{\psi_i} \in \mathbb{R}^{d_b \times d_{x^k}}$, the occurrence probability of all the state as $\hat{\mathbf{p}}^k_{\psi} \in \mathbb{R}^{d_b \times N d_{x^k}}$ and weights as $\hat{\mathbf{W}}^k \in \mathbb{R}^{d_{x^k} \times d_{x^{k+1}}}$ for $k \in \{0, \ldots, L-1\}$ and $i \in \{0, \ldots, N-1\}$.

Compute $\hat{\mathbf{x}}^L = \dfrac{\partial \mathbb{C}}{\partial \mathbf{x}^L}$ given $\mathbf{x}^L$ and $\overline{\mathbf{y}}$

$\hat{\mathbf{W}}^{L-1} = \mathbf{x}^{L-1^T} \hat{\mathbf{x}}^L$

**for** $k = L-1 : 1$ **do**

  $\hat{\mathbf{p}}^k_{\psi} = \dfrac{1}{d_{x^k}} \hat{\mathbf{x}}^{k+1} \mathbf{W}^{k^T}$

  $\hat{\mathbf{x}}^k = \sum_{i=0}^{N-1} \dfrac{(-1)^{i+1}}{N} \hat{\mathbf{p}}^k_{\psi_i}$

  $\hat{\mathbf{W}}^{k-1} = \mathbf{x}^{k-1^T} \hat{\mathbf{x}}^k$

**end**

**for** $k = 0 : L-1$ **do**

  $\mathbf{W}^k \leftarrow \text{Update}(\mathbf{W}^k, \hat{\mathbf{W}}^k, \eta)$

**end**

optimizer with a learning rate of 0.1. A total of $2^{20}$ input points were used, evenly distributed across the input space, and a batch size of $2^{10}$ was employed during training. After training, the inference computations were carried out on the same input points used during training to generate the results shown in Figure 2.2.

Table 2.1 summarizes the training and inference settings employed in the simulations. The three-layer FSM-based network used for implementing the 2D Gabor filters in Figure 2.2 achieved an MSE of approximately $1 \times 10^{-4}$ when performing the computations with a stream length of $l = 2^{15}$. To examine the effect of stream length and the number of states, we further analyzed the network with specific Gabor filter parameters, namely $\sigma^2 = 0.125$, $\gamma = 1$, $\omega = \pi/2$ and $\theta = 0^\circ$ as shown in Figure 2.2. The results demonstrated that the MSE decreases as both the stream length and the number of states increase.

**Algorithm 3:** Pseudo code of the inference computations of FSM-based networks. $L$ is the number of layers including the output layer whereas $l$ denotes the length of stochastic streams. $N$ denotes the number of states in FSMs. The Clamp function replaces the values greater than $N-1$ and less than 0 with $N-1$ and 0, respectively. The One_Hot_Encoder function converts each entry of the vector $\mathbf{s}_t$ to a one-hot encoded vector of size $N$ and concatenates the one-hot encoded vectors to form the sparse vector of $\mathbf{o}^k \in \{0,1\}^{d_b \times N d_{x^k}}$ such that $\sum_{i=j\times N}^{(j+1)\times N} o_i^k = 1$, where $o_i^k$ denotes the $i^{th}$ entry of the second dimension of the vector $\mathbf{o}^k$ for $j \in \{0, 1, \ldots, d_{x^k} - 1\}$.

**Data:** An input minibatch of $\mathbf{x}^0 \in [-1,+1]^{d_b \times d_{x^0}}$, an output minibatch of $\mathbf{y} \in [-1,+1]^{d_b \times d_{x^L}}$, the state vector of $\mathbf{s}_t^k \in \{0,\ldots,N-1\}^{d_b \times d_{x^k}}$, FSMs' output of $\mathbf{o}_t^k \in \{0,1\}^{d_b \times N d_{x^k}}$, activations of $\mathbf{x}_t^{k+1} \in \{0,1\}^{d_b \times d_{x^k}}$ and weights of $\mathbf{W}^k \in [-1,+1]^{d_{x^k} \times d_{x^{k+1}}}$ for $k \in \{0,\ldots,L-1\}$ and $i \in \{0,\ldots,N-1\}$.

$\mathbf{s}_0 = \dfrac{N}{2}$

$\mathbf{y} = 0$

**for** $t = 1 : l$ **do**

  $\mathbf{x}_t^0 = \text{Bernoulli}\left(\dfrac{\mathbf{x}^0 + 1}{2}\right)$

  **for** $k = 0 : L-1$ **do**

    $\mathbf{s}_t^k = \text{Clamp}\left(\mathbf{s}_{t-1}^k + 2 \times \mathbf{x}_t^k - 1, 0, N-1\right)$

    $\mathbf{o}_t^k = \text{One\_Hot\_Encoder}(\mathbf{s}_t^k)$

    $\mathbf{x}_t^{k+1} = \text{Bernoulli}\left(\dfrac{\mathbf{o}_t^k \text{Bernoulli}\left(\dfrac{\mathbf{W}^k + 1}{2}\right)}{d_{x^k}}\right)$

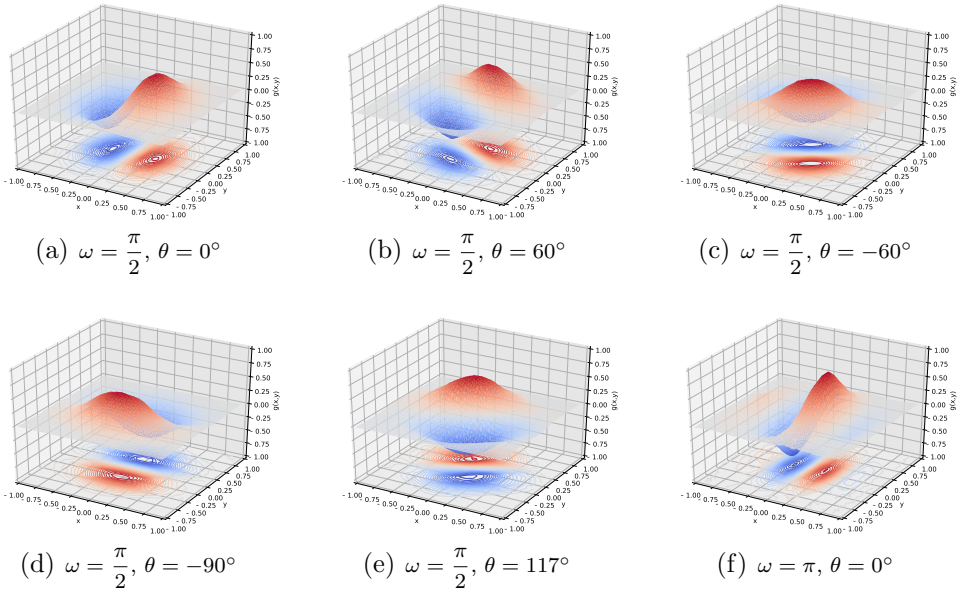
  **end**

  $\mathbf{y} = \mathbf{y} + \dfrac{2 \times \mathbf{x}_t^L - 1}{l}$

**end**

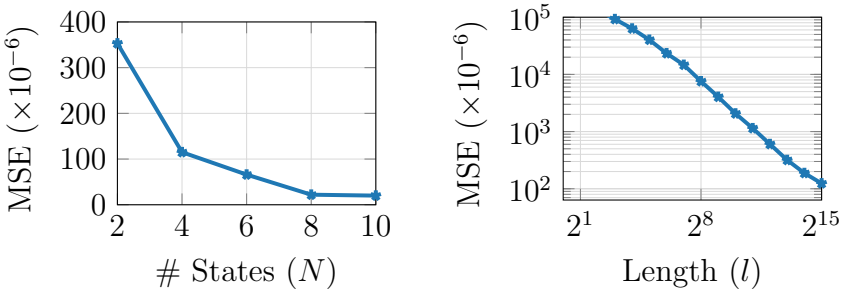
### Image Classification on MNIST Dataset

As a second application of FSM-based networks, we conduct an image classification task using the MNIST dataset of handwritten digits. The MNIST dataset consists of 60,000 grayscale images of size $28 \times 28$ pixels for training and 10,000 images for testing. For our experiments, we utilize the last 10,000 images from the training set as a validation set. To obtain the simulation results presented in Table 2.2 and Figure 2.3, we trained two three-layer FSM-based networks with configurations of 250 and 70 states (i.e., the network configurations of $784-250-250-10$ and $784-70-70-10$). These networks were trained using the



(a) $\omega = \frac{\pi}{2}$, $\theta = 0°$ (b) $\omega = \frac{\pi}{2}$, $\theta = 60°$ (c) $\omega = \frac{\pi}{2}$, $\theta = -60°$

(d) $\omega = \frac{\pi}{2}$, $\theta = -90°$ (e) $\omega = \frac{\pi}{2}$, $\theta = 117°$ (f) $\omega = \pi$, $\theta = 0°$




(g) $\omega = \frac{\pi}{2}$, $\theta = 0°$, $l = \infty$ (h) $\omega = \frac{\pi}{2}$, $\theta = 0°$, $N = 4$

Figure 2.2: (a-f) The simulation results of 2D Gabor filters with different configurations for $\sigma^2 = 0.125$ and $\gamma = 1$. (g-h) The effect of the stream length and the number of states on the performance of the 2D Gabor filters with the parameters of $\sigma^2 = 0.125$, $\gamma = 1$, $\omega = \frac{\pi}{2}$ and $\theta = 0°$.

Adam optimizer, a batch size of 100, and a learning rate of 0.1. We also applied a dropout rate of 0.15 to the hidden layers (i.e., dropping 15% of the nodes in the hidden layers) during training. Given that the primary objective of this task is to predict a label for a given image, we employed the cross-entropy (CE) loss function, which combines the cross-entropy loss with a softmax output.

The test error rates, presented in Table 2.2 and Figure 2.3, reflect the performance of our FSM-based networks. The detailed settings for training and inference in our FSM-based networks for the MNIST image classification task are provided in Table 2.3. Table 2.2 summa-

Table 2.1: The training and the inference settings used in our simulations of 2D Gabor filters.

| Simulation | FSM-Based Network | | Training Parameters | | | | | Inference Parameters |
|---|---|---|---|---|---|---|---|---|
| | Configuration | # States ($N$) | Loss ($\mathbb{C}$) | Optimizer | LR ($\eta$) | BS | # Epochs | Stream Length ($l$) |
| Figure 2.2(a-f) | $2-4-4-1$ | 4 | MSE | Adam | 0.1 | $2^{10}$ | 1000 | $2^{15}$ |
| Figure 2.2(g) | $2-4-4-1$ | 2,4,8,10 | MSE | Adam | 0.1 | $2^{10}$ | 1000 | $\infty$ |
| Figure 2.2(h) | $2-4-4-1$ | 4 | MSE | Adam | 0.1 | $2^{10}$ | 1000 | $2^1, 2^2, \ldots, 2^{15}$ |

rizes the misclassification rates of the two FSM-based networks with different configurations when performing inference computations on stochastic streams of length 128 (i.e., $l = 128$).

As shown in Table 2.2, our FSM-based networks significantly outperform existing SC-based counterparts in terms of misclassification rates and the required stream length. Additionally, our FSM-based networks require half the number of operations compared to conventional SC-based implementations of the same size. The choice of using two states and a stream length of 128 for our FSM-based networks was informed by Figure 2.3, which illustrates the misclassification rate for various numbers of states and stream lengths. As expected in the context of stochastic computing, the misclassification error decreases as the stream length increases. For stream lengths greater than 64, the error rate stabilizes, making 128 the optimal stream length, or "sweet spot." The results also indicate that the two-state FSM-based networks perform better than those with a larger number of states. This suggests that using fewer states helps to regularize the network parameters more effectively for this specific task.

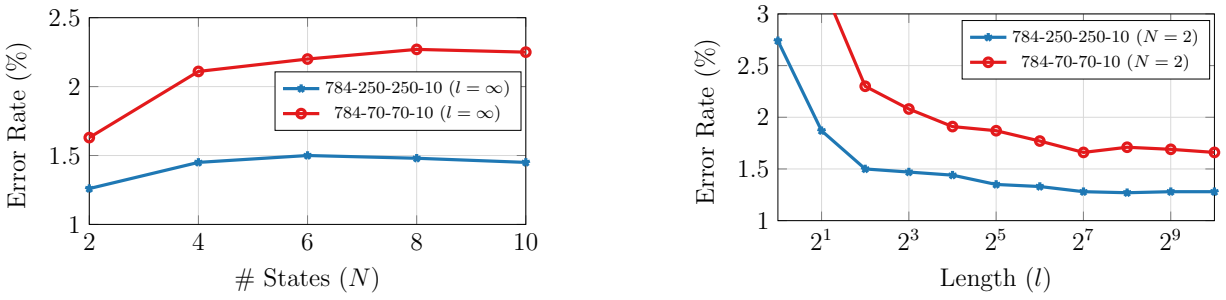



Figure 2.3: Misclassification rate of FSM-based networks on the MNIST test set using (a) different number of states and (b) different stream lengths.

Table 2.2: Performance of our FSM-based network compared to SC-based implementations on the test set of the MNIST dataset.

| Model | Configuration | ($N$, $l$) | (# Op., # Weights) | Error Rate (%) |
|---|---|---|---|---|
| FSM-based Network | 784-250-250-10 | (2, 128) | (0.52M, 0.52M) | 1.28 |
| FSM-based Network | 784-70-70-10 | (2, 128) | (0.12M, 0.12M) | 1.66 |
| TCAD'18 [48] | 784-128-128-10 | (NA, $\infty$) | (0.24M, 0.12M) | 3 |
| TCOMP'18 [17] | 784-200-100-10 | (NA, 256) | (0.36M, 0.18M) | 2.05 |
| TVLSI'17 [49] | 784-300-600-10 | (NA, 256) | (0.84M, 0.42M) | 2.01 |

## 2.6 An FSM-Based Model for Temporal Tasks

An FSM consists of three fundamental components: a transition function, an output decoder, and a memory unit [40]. The memory unit is responsible for storing the state of the machine, while the output decoder generates a sequence of outputs based on the current state, as is characteristic of Moore machines. The transition function dictates the subsequent state of the machine, taking both the present state and the current input into account. In digital systems, the memory unit is typically implemented with registers, whereas the transition function and output decoder are realized through combinational logic. Drawing inspiration from the state machine architecture used in sequential circuits, we propose an FSM-based model capable of processing temporal sequences of data. In our model, we implement both the transition function and the output decoder through a single fully-connected layer. Additionally, we utilize an FSM-based layer to serve as the memory unit. This design enables us to conceptualize the fully-connected layers and the WLFSMs as analogous to combinational logic and registers, respectively.

Table 2.3: The training and the inference settings used in our simulations to perform the image classification task on the MNIST dataset.

| Simulation | FSM-Based Network | | Training Parameters | | | | | | Inference Parameters |
|---|---|---|---|---|---|---|---|---|---|
| | Configuration | # States ($N$) | Loss ($\mathbb{C}$) | Optimizer | LR | BS | # Epochs | Dropout | Stream Length ($l$) |
| Table 2.2 | 784-250-250-10 | 2 | CE | Adam | 0.05 | 100 | 500 | 0.15 | 128 |
| Table 2.2 | 784-70-70-10 | 2 | CE | Adam | 0.05 | 100 | 500 | 0.15 | 128 |
| Figure 2.3(a) | 784-250-250-10 | $2, 4, 6, 8, 10$ | CE | Adam | 0.05 | 100 | 500 | 0.15 | $\infty$ |
| Figure 2.3(a) | 784-70-70-10 | $2, 4, 6, 8, 10$ | CE | Adam | 0.05 | 100 | 500 | 0.15 | $\infty$ |
| Figure 2.3(b) | 784-250-250-10 | 2 | CE | Adam | 0.05 | 100 | 500 | 0.15 | $2^1, 2^2, \ldots, 2^{10}$ |
| Figure 2.3(b) | 784-70-70-10 | 2 | CE | Adam | 0.05 | 100 | 500 | 0.15 | $2^1, 2^2, \ldots, 2^{10}$ |

### 2.6.1 Feed-Forward Computations

In our FSM-based model, we utilize a Moore machine, where the decision-making process depends solely on the current state. An $N$-state FSM-based model performs its feed-forward process as described by the following equations:

1. Transition Function:

$$\mathbf{z} = \mathbf{x}_t \mathbf{W}_x + \mathbf{b}_x, \tag{2.18}$$

where $\mathbf{z} \in \mathbb{R}^{d_h}$ represents the output of the transition function at time $t$, $\mathbf{x}_t$ is the input containing temporal features at time $t$, $\mathbf{W}_x \in \mathbb{R}^{d_x \times d_h}$ is the weight matrix, and $\mathbf{b}_x \in \mathbb{R}^{d_h}$ is the bias.

2. State Transition:

$$\mathbf{s}_t = \text{Clamp}\left(\mathbf{s}_{t-1} + 2 \times \text{Bernoulli}\left(\frac{\mathbf{z}+1}{2}\right) - 1, 0, N-1\right), \tag{2.19}$$

where $\mathbf{s}_t \in \{0, 1, \ldots, N-1\}^{d_h}$ represents the state values of each WLFSM in the FSM-based layer at time step $t$, The Clamp function ensures that the state values stay within the range $[0, N-1]$. Bernoulli denotes the Bernoulli distribution applied to each element of $\mathbf{z}$, producing a probabilistic state transition.

3. One-Hot Encoding:

$$\mathbf{o} = \text{One\_Hot\_Encoder}(\mathbf{s}_t), \tag{2.20}$$

where the One_Hot_Encoder function converts each entry of the vector $\mathbf{s}_t$ to a one-hot encoded vector of size $N$ and concatenates the one-hot encoded vectors to form the sparse vector of $\mathbf{o} \in \mathbb{R}^{Nd_h}$ such that $\sum_{i=j\times N}^{(j+1)\times N} o_i = 1$, where $o_i$ denotes the $i^{th}$ entry of the vector $\mathbf{o}$ for $j \in \{0, 1, \ldots, d_h - 1\}$.

4. FSM Layer Output:

$$\mathbf{q} = \text{Sigmoid}(\alpha \mathbf{o} \mathbf{W}_o + \mathbf{b}_o), \tag{2.21}$$

where $\mathbf{q} \in \mathbb{R}^{d_h}$ represents the output of the FSM-based layer. $\mathbf{b}_o \in \mathbb{R}^{d_h}$ is the bias. The parameter $\alpha$ is a fixed coefficient that prevents the weight matrix $\mathbf{W}_o \in \mathbb{R}^{Nd_h \times d_h}$ from becoming too small. For our simulations, we set $\alpha = d_h^{-1}$.

5. Output Decoder:

$$\mathbf{y} = \mathbf{q} \mathbf{W}_y + \mathbf{b}_y, \tag{2.22}$$

where $\mathbf{y} \in \mathbb{R}^{d_y}$ represents the final output of the model. The weight matrix $\mathbf{W}_y \in \mathbb{R}^{d_h \times d_y}$ and the bias $\mathbf{b}_y \in \mathbb{R}^{d_y}$ are learned during the training phase.

This FSM-based model integrates the classical state machine architecture with modern deep learning elements to process temporal data effectively. The memory unit, transition function, and output decoder work together to perform sequence processing, where the state transitions depend on both the input and previous states, and the output is determined solely by the current state.

### 2.6.2 Backpropagation

The primary challenge in training the FSM-based model lies in computing the derivative of the vector $\mathbf{o}$ with respect to $\mathbf{z}$. In contrast, the gradients of other computations, such as the matrix-vector multiplications in Equation (2.18), Equation (2.21), and Equation (2.22), can be efficiently derived using the chain rule. The purpose of employing one-hot encoded vectors, as defined in Equation (2.20), is to ensure that only the weights corresponding to the current state of the WLFSMs are activated. Each selected weight corresponds to either a forward transition (i.e., the transition from $\psi_{i-1}$ to $\psi_i$) when the Bernoulli function outputs 1, or a backward transition (i.e., the transition from $\psi_i$ to $\psi_{i-1}$) when the Bernoulli function outputs 0. Here, $\psi_i$ denotes the $i^{th}$ state of a WLFSM with $N$ states, where $i \in \{0, 1, \ldots, N-1\}$. The probability of selecting weights associated with state $\psi_i$ during a forward transition is denoted as $p_z$, and for a backward transition, it is $1 - p_z$. This probability is determined by the input $z \in [-1, 1]$ of the WLFSM, where $p_z$ is defined as $p_z = \frac{1+z}{2}$. Based on these probabilities, the gradient of the state vector $\mathbf{s}_t$ with respect $\mathbf{z}$ is expressed as:

$$\frac{\partial \mathbf{s}_t}{\partial \mathbf{z}} = \begin{cases} 1 & \text{when Bernoulli}\left(\dfrac{\mathbf{z}+1}{2}\right) == 1 \\ -1 & \text{otherwise} \end{cases}. \tag{2.23}$$

During the backpropagation process through the One_Hot_Encoder function, only the gradients corresponding to the current state of the WLFSMs are propagated. This is mathematically represented as:

$$\hat{s}_{t_j} = \sum_{i=j\times N}^{(j+1)\times N} (o_i \times \hat{o}_i), \tag{2.24}$$

where $\hat{s}_{t_j}$ is the $j^{th}$ entry of the gradient vector $\hat{\mathbf{s}}_t \in \mathbb{R}^{d_h}$ at the input of the One_Hot_Encoder function for $j \in \{0, 1, \ldots, d_h - 1\}$ and $\hat{o}_i$ the $i^{th}$ entry of the gradient vector $\hat{\mathbf{o}} \in \mathbb{R}^{Nd_h}$ at the

output of the One_Hot_Encoder function for $j \in \{0, 1, \ldots, Nd_h - 1\}$.

### 2.6.3 Training Details

Here, we present the training method for our FSM-based models. Prior to training for a specified number of time steps $l$, the state values of the FSMs are initialized to $\lfloor N/2 \rfloor$. The transition function and the output decoder execute fully connected computations as described in Equation (2.18) and Equation (2.22), respectively. Within the memory unit (i.e., the FSM-based layer), the state values are either incremented or decremented by stochastically sampling inputs to this layer, as defined in Equation (2.19). During the training procedure of FSM-based models, the Bernoulli function with a fixed seed must be employed in both forward propagation (see Equation (2.19)) and backward propagation (see Equation (2.23)). The use of a fixed seed ensures that the transition directions in forward propagation remain consistent during backward propagation, which is critical for correct gradient computation. The detailed training method is provided in Algorithm 4.

As discussed in Section 2.6, gradients are backpropagated, and parameters are updated at the end of each time step in FSM-based models (see Algorithm 4). It is important to note that the state values can also be updated deterministically. Specifically, the Sign function can replace the Bernoulli function in a deterministic approach. Both the stochastic and deterministic approaches yield the same accuracy performance; however, the deterministic approach, which uses the Sign function, results in faster training due to its lower computational cost compared to the Bernoulli function.

### 2.6.4 Simulation Results

As discussed earlier, the states of our FSM-based model are updated based solely on the present input. Specifically, the transition function increments or decrements the state of each Weighted Linear Finite State Machine (WLFSM) according to the input features at time $t$. Consequently, the FSM-based model can be interpreted as a time-homogeneous process, where the probability of state transitions remains independent of $t$.

For temporal tasks requiring decisions at each time step (e.g., the CLLM task), backpropagation in the FSM-based model is performed at the end of each time step. This approach significantly reduces the storage required for intermediate values during training by a factor of $l\times$, enabling the FSM-based model to process extremely long data sequences efficiently. This is in stark contrast to LSTM networks, where the network must be unrolled across all time steps, and backpropagation is applied to the entire unrolled network. Consequently,

**Algorithm 4:** Pseudo code of the training algorithm for FSM-based models. $l$ is the number of time steps. The training loss is denoted as $\mathbb{C}$. $N$ and $\eta$ denote the number of states in FSMs and the learning rate, respectively. The gradient of parameters w.r.t. $\mathbb{C}$ is denoted by " $\hat{}$ " over their corresponding symbols. The Clamp function replaces the values greater than $N-1/1$ and less than $0/-1$ with $N-1/1$ and $0/-1$, respectively. $\hat{\sigma}$ denotes the derivative of the Sigmoid function. The One_Hot_Encoder function converts each entry of the vector $\mathbf{s}_t$ to a one-hot encoded vector of size $N$ and concatenates the one-hot encoded vectors to form the sparse vector of $\mathbf{o} \in \{0,1\}^{d_b \times N d_h}$ such that $\sum_{i=j\times N}^{(j+1)\times N} o_i = 1$, where $o_i$ denotes the $i^{th}$ entry of the second dimension of the vector $\mathbf{o}$ for $j \in \{0,1,\ldots,d_h-1\}$. The parameter $\alpha$ is set to $d_h^{-1}$. "$\times$" denotes element-wise multiplications. Note that $d_x$ is equal to $d_y$ in the CLLM task.

**Data:** An input minibatch of $\mathbf{X} \in \mathbb{N}^{d_b \times d_x \times l}$, an input minibatch of $\mathbf{x}_t \in \mathbb{N}^{d_b \times d_x}$ at the time step $t$, a target minibatch of $\overline{\mathbf{Y}} \in \mathbb{N}^{d_b \times d_x \times l}$, a target minibatch of $\overline{\mathbf{y}}_t \in \mathbb{N}^{d_b \times d_x}$ at the time step $t$, the transition function's output $\mathbf{z} \in [-1,1]^{d_b \times d_h}$, the transition function's weights $\mathbf{W}_x \in \mathbb{R}^{d_x \times d_h}$, the transition function's biases $\mathbf{b}_x \in \mathbb{R}^{d_h}$, the state vector of $\mathbf{s}_t \in \{0,\ldots,N-1\}^{d_b \times d_h}$, the FSMs' output of $\mathbf{o} \in \{0,1\}^{d_b \times N d_h}$, the FSM-based layer's output of $\mathbf{q} \in \mathbb{R}^{d_b \times d_h}$, the FSM-based layer's weights $\mathbf{W}_o \in \mathbb{R}^{N d_h \times d_h}$, the FSM-based layer's biases $\mathbf{b}_o \in \mathbb{R}^{d_h}$, the output decoder's output $\mathbf{y} \in \mathbb{R}^{d_b \times d_y}$, the output decoder's weights $\mathbf{W}_y \in \mathbb{R}^{d_h \times d_y}$, the output decoder's biases $\mathbf{b}_y \in \mathbb{R}^{d_y}$, the gradient of the transition function's output $\hat{\mathbf{z}} \in \mathbb{R}^{d_b \times d_h}$, the gradient of the transition function's weights $\hat{\mathbf{W}}_x \in \mathbb{R}^{d_x \times d_h}$, the gradient of the transition function's biases $\hat{\mathbf{b}}_x \in \mathbb{R}^{d_h}$, the gradient of the state vector $\hat{\mathbf{s}}_t \in \{0,\ldots,N-1\}^{d_b \times d_h}$, the gradient of the FSMs' output $\hat{\mathbf{o}} \in \{0,1\}^{d_b \times N d_h}$, the gradient of the FSM-based layer's output $\hat{\mathbf{q}} \in \mathbb{R}^{d_b \times d_h}$, the gradient of the FSM-based layer's weights $\hat{\mathbf{W}}_o \in \mathbb{R}^{N d_h \times d_h}$, the gradient of the FSM-based layer's biases $\hat{\mathbf{b}}_o \in \mathbb{R}^{d_h}$, the gradient of the output decoder's output $\hat{\mathbf{y}} \in \mathbb{R}^{d_b \times d_y}$, the gradient of the output decoder's weights $\hat{\mathbf{W}}_y \in \mathbb{R}^{d_h \times d_y}$ and the gradient of the output decoder's biases $\hat{\mathbf{b}}_y \in \mathbb{R}^{d_y}$ for $t \in \{1,\ldots,l\}$.

$\mathbf{s}_0 = \lfloor \frac{N}{2} \rfloor$
**for** $t = 1 : l$ **do**
  $\mathbf{x}_t = \mathbf{X}[:,:,t]$
  $\overline{\mathbf{y}} = \overline{\mathbf{Y}}[:,:,t]$
  $\mathbf{z} = \text{Clamp}\left(\mathbf{x}_t \mathbf{W}_x + \mathbf{b}_x, -1, 1\right)$
  $\mathbf{s}_t = \text{Clamp}\left(\mathbf{s}_{t-1} + 2 \times \text{Bernoulli}\left(\frac{\mathbf{z}+1}{2}\right) - 1, 0, N-1\right)$
  $\mathbf{o} = \text{One\_Hot\_Encoder}(\mathbf{s}_t)$
  $\mathbf{q} = \text{Sigmoid}(\alpha \mathbf{o} \mathbf{W}_o + \mathbf{b}_o)$
  $\mathbf{y} = \mathbf{q}\mathbf{W}_y + \mathbf{b}_y$
  $\mathbf{h} = \text{Softmax}(\mathbf{y})$
  $\mathbb{C} = \text{Cross\_Entropy}(\mathbf{h}, \overline{\mathbf{y}}_t)$
  $\hat{\mathbf{y}} = \frac{\partial \mathbb{C}}{\partial \mathbf{y}} = \overline{\mathbf{y}}_t - \mathbf{h}$
  $\hat{\mathbf{q}} = \hat{\mathbf{y}} \mathbf{W}_y^T$
  $\hat{\mathbf{W}}_y = \mathbf{q}^T \hat{\mathbf{y}}$
  $\hat{\mathbf{o}} = \alpha(\hat{\sigma}(\alpha \mathbf{o} \mathbf{W}_o + \mathbf{b}_o) \times \hat{\mathbf{q}}) \mathbf{W}_o^T$
  $\hat{\mathbf{W}}_o = \alpha \mathbf{o}^T (\hat{\sigma}(\alpha \mathbf{o} \mathbf{W}_o + \mathbf{b}_o) \times \hat{\mathbf{q}})$
  $\hat{s}_{t_j} = \sum_{i=j\times N}^{(j+1)\times N} (o_i \times \hat{o}_i)$
  $\hat{\mathbf{z}} = \hat{\mathbf{s}}_t \times \left(2 \times \text{Bernoulli}\left(\frac{\mathbf{z}+1}{2}\right) - 1\right)$
  $\hat{\mathbf{W}}_x = \mathbf{x}_t^T \hat{\mathbf{z}}$
  $\mathbf{W}_y \leftarrow \text{Update}(\mathbf{W}_y, \hat{\mathbf{W}}_y, \eta)$
  $\mathbf{b}_y \leftarrow \text{Update}(\mathbf{b}_y, \hat{\mathbf{y}}, \eta)$
  $\mathbf{W}_o \leftarrow \text{Update}(\mathbf{W}_o, \hat{\mathbf{W}}_o, \eta)$
  $\mathbf{b}_o \leftarrow \text{Update}(\mathbf{b}_o, \hat{\sigma}(\alpha \mathbf{o} \mathbf{W}_o + \mathbf{b}_o) \times \hat{\mathbf{q}}, \eta)$
  $\mathbf{W}_x \leftarrow \text{Update}(\mathbf{W}_x, \hat{\mathbf{W}}_x, \eta)$
  $\mathbf{b}_x \leftarrow \text{Update}(\mathbf{b}_x, \hat{\mathbf{z}}, \eta)$
**end**

the sequence length that can be processed by LSTMs during training is typically limited to a few hundred time steps, as the storage required to retain intermediate values for the unrolled network can easily exceed the memory capacity of modern GPUs. For instance, Figure 2.4 illustrates the memory usage and test accuracy of the LSTM and FSM-based models on the GeForce GTX 1080 Ti for varying time steps, with both models configured to have the same number of weights and a batch size of 100 for the CLLM task on the Penn Treebank dataset [50]. The results indicate that the memory usage of the FSM-based model is independent of the number of time steps, making it highly suitable for on-chip learning in mobile devices with limited storage. By contrast, although the LSTM model demonstrates a slight improvement in performance, it becomes infeasible to train beyond 2000 time steps due to excessive memory requirements.

In addition to its lower memory footprint, the FSM-based model is less computationally intensive. Specifically, the backward process in the FSM-based model requires computations only for the current time step, whereas the LSTM model must compute gradients for all unrolled time steps. This reduction in both memory and computational requirements directly translates to lower power consumption. For instance, our measurements using the NVIDIA System Management Interface show that training FSM-based models of size 1000 with a batch size of 100 draws approximately 160W across time steps ranging from 100 to 2500. In contrast, training LSTM models of the same size consumes between 205W and 245W. Finally, increasing the number of time steps significantly impacts the convergence rate of LSTM models, as shown in Figure 2.4. In contrast, the convergence rate of the FSM-based model remains unaffected, further highlighting the efficiency and scalability of the proposed approach.

To demonstrate the effectiveness of our FSM-based model in processing temporal data, we evaluated its performance on the CLLM task using the Penn Treebank [50], War & Peace [51], and Linux Kernel [51] corpora. The performance is measured in terms of bits per character (BPC). The simulation results for our FSM-based model are summarized in Table 2.4

According to the experimental results, our FSM-based model with 4-state FSMs achieves

Table 2.4: Performance of our FSM-based model on the CLLM task.

| Model | Penn Treebank | | | War & Peace | | | Linux Kernel | | |
|---|---|---|---|---|---|---|---|---|---|
| | # Weights | # Op. | BPC | # Weights | # Op. | BPC | # Weights | # Op. | BPC |
| 4-State FSM-based Model | 4.1M | 1.1M | 1.52 | 1.1M | 0.3M | 1.89 | 1.1M | 0.3M | 1.93 |
| LSTM (Our implementation) | 4.1M | 8.1M | 1.45 | 1.1M | 2.1M | 1.83 | 1.1M | 2.1M | 1.85 |

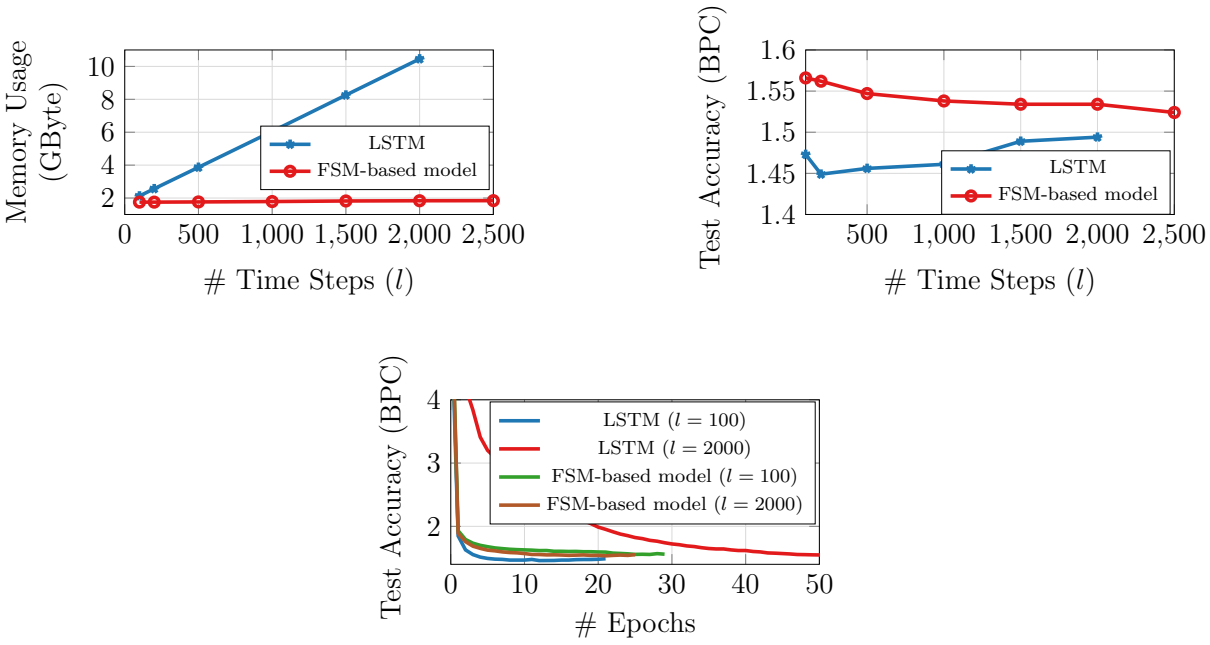



Figure 2.4: The memory usage and the test accuracy performance of an LSTM model with 1000 hidden states versus a 4-state FSM-based mode of size 1000 (i.e., $d_h = 1000$) for different numbers of time steps and training epochs when performing the CLLM task on the Penn Treebank corpus.

accuracy comparable to that of the LSTM model in terms of BPC when both models have the same number of parameters. To ensure a fair comparison, we set the number of hidden nodes for all models to $d_h = 1000$ for the Penn Treebank corpus and $d_h = 1000$ for the War & Peace and Linux Kernel corpora, as reflected in Table 2.4. Notably, our FSM-based network requires only $\frac{1}{7}$ the number of operations compared to an LSTM model of the same size. This efficiency stems from the computational simplicity of the WLFSM layer, where operations are limited to indexing and accumulation, unlike the more complex operations required in LSTMs. These findings highlight the computational advantages of our FSM-based model while maintaining competitive performance for temporal tasks.

### 2.6.5 Training Settings

In this section, we present the training settings and model architectures employed to obtain the results reported in Table 2.4 and Figure 2.4. The Character-Level Language Modeling (CLLM) task was conducted using our FSM-based model on three corpora: Penn Treebank (PT), War & Peace (WP), and Linux Kernel (LK).

Table 2.5: The training settings used in our simulations to perform the CLLM task on the Penn Treebank, War & Peace and Linux Kernel datasets.

| Simulation/Dataset | Network Configuration | | | Training Parameters | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Model | # States ($N$) | Size ($d_h$) | Loss ($\mathbb{C}$) | Optimizer | LR ($\eta$) | BS | # Epochs | Dropout | Time Step ($l$) |
| Figure 2.4 (left subfigure)/PT | FSM-based | 4 | 1000 | Cross-Entropy | Adam | 0.05 | 100 | 500 | 0.15 | $100 - 2500$ |
| Figure 2.4 (left subfigure)/PT | LSTM | NA | 1000 | Cross-Entropy | Adam | 0.001 | 100 | 500 | 0 | $100 - 2000$ |
| Figure 2.4 (middle subfigure)/PT | FSM-based | 4 | 1000 | Cross-Entropy | Adam | 0.05 | 100 | 500 | 0.15 | $100 - 2500$ |
| Figure 2.4 (middle subfigure)/PT | LSTM | NA | 1000 | Cross-Entropy | Adam | 0.001 | 100 | 500 | 0 | $100 - 2000$ |
| Figure 2.4 (right subfigure)/PT | FSM-based | 4 | 1000 | Cross-Entropy | Adam | 0.05 | 100 | 29 | 0.15 | 100 |
| Figure 2.4 (right subfigure)/PT | FSM-based | 4 | 1000 | Cross-Entropy | Adam | 0.05 | 100 | 25 | 0.15 | 2000 |
| Figure 2.4 (right subfigure)/PT | LSTM | NA | 1000 | Cross-Entropy | Adam | 0.001 | 100 | 21 | 0 | 100 |
| Figure 2.4 (right subfigure)/PT | LSTM | NA | 1000 | Cross-Entropy | Adam | 0.001 | 100 | 50 | 0 | 2000 |
| Table 2.4/PT | FSM-based | 4 | 1000 | Cross-Entropy | Adam | 0.05 | 100 | 500 | 0.15 | 2500 |
| Table 2.4/PT | LSTM | NA | 1000 | Cross-Entropy | Adam | 0.001 | 100 | 500 | 0 | 100 |
| Table 2.4/WP | FSM-based | 4 | 500 | Cross-Entropy | Adam | 0.05 | 100 | 500 | 0.15 | 2000 |
| Table 2.4/WP | LSTM | NA | 500 | Cross-Entropy | Adam | 0.001 | 100 | 500 | 0 | 100 |
| Table 2.4/LK | FSM-based | 4 | 500 | Cross-Entropy | Adam | 0.05 | 100 | 500 | 0.15 | 2000 |
| Table 2.4/LK | LSTM | NA | 500 | Cross-Entropy | Adam | 0.001 | 100 | 500 | 0 | 100 |

**Penn Treebank**: The Penn Treebank corpus was divided into training, validation, and test sets comprising 5017k, 393k, and 442k characters, respectively, with a character vocabulary size of 50. For this task, we used an FSM-based model with 1000 hidden units (i.e., $d_h = 1000$). The cross-entropy loss was minimized using the ADAM optimization algorithm with a learning rate of 0.05. Training was performed in mini-batches of size 100.

**Linux Kernel and War & Peace**: The Linux Kernel and Leo Tolstoy's War & Peace corpora consist of 6,206,996 and 3,258,246 characters, respectively, with character vocabularies of size 101 and 87. The Linux Kernel corpus was split into 4566k, 621k, and 621k characters for the training, validation, and test sets, respectively, while the War & Peace corpus was divided into 2932k, 163k, and 163k characters. For both corpora, we used an FSM-based model with 500 hidden units (i.e., $d_h = 500$). The cross-entropy loss was minimized using the ADAM optimizer with a learning rate of 0.05, and training was performed using mini-batches of size 100.

The training procedure followed the method described in Algorithm 4. Table 2.5 summarizes the training settings used for the results presented in Table 2.4 and Figure 2.4. Notably, a dropout rate of 0.15 was applied to the final layer of our FSM-based networks during training, meaning 15% of the output decoder's nodes were dropped. Additionally, we used the number of time steps (i.e., sequence length $l$) that yielded the best bits-per-character (BPC) performance for both the LSTM and FSM-based models as reported in Table 2.4.

## 2.7 Conclusion

In this chapter, we introduced a novel method for training WLFSMs, which are computational models capable of processing sequential data. To enable WLFSMs to perform non-sequential

tasks, we employed SC to convert continuous values into stochastic bit streams. Networks composed solely of WLFSMs that perform computations on these stochastic bit streams are referred to as FSM-based networks.

As a first application of FSM-based networks, we implemented 2D Gabor filters using only 10 WLFSMs, each with 4 states. In [52], a SC-based implementation of 2D Gabor filters was introduced, where the sinusoidal component in Equation (2.17) was approximated using multiple tanh functions. This method involved using a 256-state FSM for the exponential part and 9 56-state FSMs for the sinusoidal part, with a stream length of $2^{18}$, to achieve a similar MSE to that obtained in our simulations. However, this approach is limited to functions that can be approximated by tanh or exponential functions. In contrast, our FSM-based network provides a more general solution that can implement any arbitrary target function.

For the second application, we applied FSM-based networks to a classification task on the MNIST dataset and demonstrated that our FSM-based networks significantly outperform conventional SC-based implementations in terms of both misclassification error and the number of operations.

Finally, as a major contribution of this work, we introduced an FSM-based model capable of performing temporal tasks. We showed that, unlike LSTMs, the required storage for training our FSM-based models is independent of the number of time steps. This property allows our FSM-based models to learn extremely long data dependencies while achieving substantial resource savings: reducing the storage required for intermediate training values by a factor of $l\times$, lowering the power consumption during training by 33%, and decreasing the number of operations during inference by a factor of $7\times$.

# 3

# Dynamic Sign-Magnitude Stochastic Representation for Stochastic Computing

In this work, we introduce Dynamic Sign-Magnitude (DSM), a novel representation of stochastic streams that combines multiple stochastic representations to improve computational accuracy in the stochastic computing (SC) domain. DSM leverages previously proposed stochastic representations and incorporates them into our SC-based neural network (NN) framework. To facilitate this, we propose a new binary multiplication method capable of performing ternary (i.e., $-1, 0, 1$) operations for the combined SC representations. Additionally, we present a training methodology designed specifically for SC-based NNs, utilizing only binary/ternary operations and adders. In particular, we estimate the polarity of gradients using ternary values during back-propagation. Our experimental results demonstrate that training SC-based NNs using the proposed method significantly reduces computational latency while achieving comparable accuracy to state-of-the-art approaches. To the best of our knowledge, this is the first work that replaces all full-precision computations required for both forward and backward propagation with simple binary/ternary operations, without introducing additional

circuit complexity—unlike the SC-based NN in [17]. The main contributions of this work are summarized as follows:

- **Dynamic Sign-Magnitude Stochastic Representation**: We propose a novel stochastic stream representation to enhance computational accuracy in the SC domain. By integrating multiple stochastic streams, we introduce a binary multiplication method that replaces ternary operations with binary operations during forward propagation. This new representation enables more accurate computations using shorter sequence lengths compared to existing SC-based NNs.

- **Efficient Back-Propagation Training**: We propose a new training algorithm for SC-based NNs that employs ternary operations exclusively for back-propagation. The design eliminates the need for full-precision multipliers, replacing them with simple binary/ternary operations to enable efficient and effective training of SC-based NNs.

## 3.1 Disclaimer

The content of this chapter is adapted and expanded from our prior work published in the *IEEE Design & Test* journal ©IEEE 2021 [1], in a paper titled *"Training Binarized Neural Networks Using Ternary Multipliers"*, authored by Amir Ardakani, Arash Ardakani, and Warren J. Gross. I developed the algorithms and conducted the experiments. Dr. Arash Ardakani and Professor Warren J. Gross provided valuable suggestions to enhance the experimental framework and contributed to the revision of the original published work. In this chapter, I have included a revised and extended version of this work. The text has been restructured and refined to more clearly articulate our contributions. Additionally, the chapter expands on the original publication by providing a detailed mathematical explanation, accompanied by an illustrative example, to further clarify the proposed dynamic signed-magnitude stochastic representation (see Section 3.4.2).

## 3.2 Related Work

Numerous studies have explored the binarization of deep neural networks (DNNs) using stochastic computing (SC), demonstrating comparable performance to conventional binary-radix approaches [17,36,48,49,53]. SC-based neural networks (SC-based NNs) are particularly advantageous for low-cost implementations, offering significant reductions in power consumption and resource utilization compared to traditional binary-radix designs of DNNs [49].

Additionally, SC-based implementations exhibit higher resilience to soft errors, such as bit flips, making them ideal for robust computing systems.

Given these advantages, SC-based NNs are well-suited for application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) implementations, particularly in scenarios where power efficiency and resource optimization are critical. Consequently, SC has been utilized to implement a wide range of DNN architectures, including spiking neural networks, fully connected neural networks (FCNNs), restricted Boltzmann machines (RBMs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs) [36,48]. However, these SC-based implementations have predominantly been used for the inference stage, where network parameters are obtained from pre-trained models. The challenge of enabling efficient on-chip learning for SC-based NNs remains an open research problem.

Ardakani et. al., proposed integral representation of stochastic streams and designed a new FSM-based *Stanh* function that takes integral stochastic streams and outputs binary stochastic streams in return [49]. They manged to produce comparable results on MNIST dataset using short sequence lengths (e.g., 16) and lower area and energy consumption of SC-based system compared to its binary-radix counterpart at the cost of larger latency (i.e., 5/3× slower). Li et. al., proposed an efficient area/power SC-based architecture which exploits approximate parallel counter-based and multiplexer-based neurons [54]. Their design achieves 55× and 151× improvement in terms of area and power, respectively, compared with a conventional binary-radix implementation. Liu et al. [17] were among the first to propose a method for training FCNNs using stochastic computing. Their approach performs both training and inference computations on stochastic streams, marking a significant step toward realizing fully SC-based neural networks. Despite the promising results, their method requires long sequence length (sequence length of size 256 with 16× parallelization) to achieve accuracy comparable to state-of-the-art models, which can limit its practicality. In addition, they proposed a reconfigurable stochastic computational activation unit to implement different types of activation functions such as *tanh* and *ReLU* functions. They managed to achieve a 2× improvement in terms of area compared to a fixed-point implementation at the cost of a very small drop of accuracy (i.e., a 0.15-0.33% accuracy drop). Liu et al. [48] subsequently proposed a method that partially performs back-propagation computations using binary operations by stochastically sampling from full-precision gradients to update weights. While this method simplifies certain computations by using binary operations, local gradients are still computed in real value, leaving room for further optimization.

The accuracy of the stochastic stream significantly falls short in representing near-zero

values with bipolar representation of SC. This becomes an important issue when NNs are implemented with SC as a large number of the weights in NNs are near-zero values. Zhakatayev et. al., proposed sign-magnitude stochastic stream to address this issue [55]. They were able to use sequence lengths as short as 32 and to produce state-of-the-art results in SC domain.

## 3.3 Preliminaries

### 3.3.1 Stochastic Computing and Circuits

Stochastic Computing (SC) operates on stochastic streams, which are sequences of random bits that represent continuous values. Let $x_t \in \{0, 1\}$ for $t \in \{0, 1, \ldots, N\}$ denote a random binary variable in the stochastic stream $\mathbf{X}$, where $N$ is the sequence length. In the *unipolar* representation, a real number $x \in [0, 1]$ is encoded such that $\mathbb{E}[x_t]$ equals to $x$. For a sufficiently long bit sequence, the mean value of the sequence is expected to converge to the theoretical expectation [16], i.e., $\frac{1}{N}\sum_{t=1}^{N} x_t = \mathbb{E}[x_t] = x$. Stochastic streams can also represent negative numbers using the *bipolar* format. Here, a real number $x \in [-1, 1]$ is encoded such that the $\mathbb{E}[x_t]$ equals to $\frac{x+1}{2}$. Any real number outside the $[-1, 1]$ range can also be represented by scaling it to fit within the appropriate unipolar or bipolar range. Stochastic streams are typically generated in hardware using linear-feedback shift registers (LFSRs) and comparators. For software simulations, they can be created by sampling from the *Bernoulli* distribution [36]. In SC, arithmetic operations are implemented using basic logic gates. For example, multiplications are performed with AND gates in unipolar format and XNOR gates in bipolar format. Stochastic additions, on the other hand, can be performed with either OR gates or scaled adders. An OR gate is suitable for addition when the input values are small [49], while a scaled adder can be implemented as a two-input multiplexer (MUX). The MUX's selector is controlled by a stochastic stream with a probability of 0.5, resulting in an output with a probability equal to the average probability of the input. To perform addition with $M$ inputs, a tree of two-input MUXs is used, requiring longer stochastic streams to compensate for the precision loss caused by $M\times$ downscaling [49].

A recent innovation in SC is the *Integral* stochastic stream, proposed as an alternative to the conventional unipolar and bipolar formats to simplify addition and enhance performance in SC-based DNN implementations [49]. An integral stochastic stream represents a sequence of integers and can be generated by summing multiple unipolar or bipolar stochastic streams. When generating integral streams, zeros in bipolar streams are treated as -1. The average value of an integral sequence corresponds to its real value. For instance, 1.25 can be represented

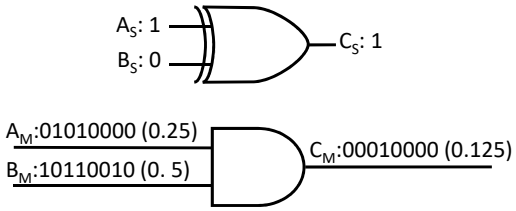



Figure 3.1: Multiplication between two SM streams ($a = -0.25$ and $b = 0.5$) with sequence length of 8. $A_S/B_S$ and $A_M/B_M$ denote the sign and magnitude of the stochastic streams $A/B$, respectively. ©IEEE 2021 [1].

as {1,1,1,0,1,0,1,1} (0.5) + {1,1,1,0,1,1,1,1} (0.75) in bipolar and {2,2,2,-2,2,0,2,2} (1.25) in integral formats. This work employs the integral format for representing the inputs of non-linear activation functions.

The Sign-Magnitude (SM) format is another stochastic representation introduced to enhance computational accuracy in neural networks [55]. To represent a real number $x \in [-1, +1]$ using SM, the absolute value of $x$ is encoded in unipolar format while preserving its sign for subsequent computations. The bit-width of SM streams is thus one bit longer than unipolar and bipolar formats. Multiplying two SM-formatted values requires two gate operations: AND and XOR gates compute the magnitude and sign of the product, respectively, as illustrated in Figure 3.1. In Section 3.4, we detail how our proposed SC-based neural network leverages the SM format while performing multiplications using only a single gate.

For a comprehensive background on SC and its foundational principles, please refer to Appendix A.

### 3.3.2 Back-propagation: Gradient Descent

Gradient descent (GD) is one of the most widely used algorithms for training deep neural networks. For DNNs with a differentiable loss function $\mathbb{L}$, , the local gradient of the $j^{th}$ neuron in $\ell^{th}$ layer is given by:

$$g_j^{(\ell)} = \begin{cases} \frac{\partial \mathbb{L}}{\partial f(v_j^{(\ell)})} & \text{output layer} \\ f'(v_j^{(\ell)}) \sum_m g_j^{(\ell+1)} w_{m,j}^{(\ell)} & \text{middle layers} \end{cases}, \tag{3.1}$$

where $g_j^{(\ell)}$ is the local gradient of the $j^{th}$ neuron in $\ell^{th}$ layer, $f$ is the non-linear activation function, and $w_{m,j}^{(\ell)}$ is the weight connecting the $m^{th}$ neuron in the $\ell + 1^{th}$ layer to the $j^{th}$ neuron in the $\ell^{th}$ layer. The variable $v_j^{(\ell)}$ represents the weighted sum at the $j^{th}$ neuron in

the $\ell^{th}$ layer and is computed as:

$$v_j^{(\ell)} = \sum_k w_{j,k}^{(\ell-1)} \ f(v_j^{(\ell-1)}). \tag{3.2}$$

Note that in the first layer,$f(v_j^{(\ell-1)})$ is substituted by the input feature values $X$ for the first layer. The gradients with respect to the weights are calculated as:

$$\nabla w_{j,k}^{(\ell)} = g_j^{(\ell)} \ f(v_j^{(\ell-1)}). \tag{3.3}$$

The weights are then updated using the current weights $w^t$, the learning rate $\alpha$ and the gradients obtained from Eq. (3.3) as follows:

$$w^{t+1} = w^t - \alpha \nabla w. \tag{3.4}$$

It is worth noting that the matrix multiplications in Equation (3.1), Equation (3.2), and Equation (3.3) can be replaced with convolution operations when performing back-propagation in convolutional neural networks.

## 3.4 Stochastic Neural Networks

This section outlines the algorithms and functions necessary to facilitate the forward and backward propagation of the proposed SC-based NN using binary and ternary operations. First, we describe the SC-based implementation of the non-linear activation function commonly utilized in DNNs. Next, we provide a detailed explanation of the binarization process applied to both the nodes and the parameters of the network. Finally, we present the algorithm developed to execute the forward and backward propagation steps efficiently.

### 3.4.1 Integral Stochastic tanh Activation Function

The implementation of complex functions in stochastic computing (SC) presents significant challenges. Typically, such functions are realized using (FSMs) [49]. FSMs are designed as up-down saturating counters, where the output is determined by comparing the counter's value with a predefined threshold. In SC-based neural network implementations, the input to the non-linear activation function is generally generated by scaled adders. However, scaled adders reduce the precision of stochastic streams when handling a large number of inputs (as discussed in Subsection 3.3.1). To address this limitation, we adopt the integral SC adder and its corresponding FSM-based function, referred to as *IStanh*, in this work.

The *IStanh* function is implemented using FSMs, following a design similar to the conventional *Stanh* architecture [49]. However, unlike the conventional *Stanh*, the *IStanh* function can take multiple steps to transition between states, depending on its integral stochastic input. The output of *IStanh* is binary (0 or 1), producing a bipolar stochastic stream based on its current state. IStanh function is covered in-depth in Appendix A

### 3.4.2 Dynamic Sign-Magnitude Stochastic stream

In neural networks, weights are typically initialized with values near zero [48]. Moreover, even after the training phase, weights can retain near-zero values. As demonstrated in [48], the bipolar format struggles to accurately represent these near-zero values, leading to a decrease in accuracy in stochastic computations. To address this limitation, we adopt the Sign-Magnitude representation of stochastic streams, as suggested in [55]. However, in our approach, the SM format is exclusively used to represent the values of weights, while the values of nodes (i.e., activations) are represented in bipolar format. It is worth emphasizing that the activations are generated by the $IStanh$ function, which inherently produces a bipolar stochastic stream. This hybrid representation allows for improved accuracy while leveraging the strengths of both formats in stochastic computations.

To enable multiplication between bipolar and SM stochastic streams, we propose employing a single XNOR gate, as illustrated in Figure 3.2.

Let $b_t \in \{0, 1\}$ be a random binary variable within the bipolar stream $\mathbf{B}$, which is encoded to represent the real value $b \in [-1, 1]$, such that:

$$\mathbb{E}[b_t] = \frac{b+1}{2},$$

$$2\mathbb{E}[b_t] - 1 = \mathbb{E}[2b_t - 1] = b. \tag{3.5}$$

By substituting $2b_t - 1$ with its equivalent $(-1)^{1-b_t}$, the following expression is obtained:

$$\mathbb{E}[(-1)^{1-b_t}] = b. \tag{3.6}$$

Similarly, let $m_t \in \{0, 1\}$ be a random binary variable within the SM stream $\mathbf{M}$, which is encoded to represent the real value $m \in [-1, 1]$, such that:

$$\mathbb{E}[m_t] = |m|,$$

$$(-1)^s \cdot \mathbb{E}[m_t] = \mathbb{E}[(-1)^s \cdot m_t] = m, \tag{3.7}$$

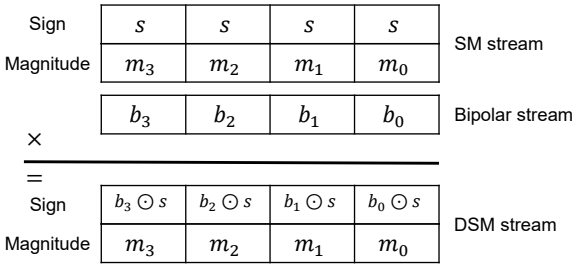



Figure 3.2: Multiplication between SM and bipolar streams with sequence length of 4. The sign value of the SM stream is used in every multiplications between the elements in the magnitude of the SM stream and the elements of the bipolar stream. The result of this multiplication is considered as DSM stream that has a sign value for each element of the sequence. ©IEEE 2021 [1].

where $|m|$ is the absolute value of $m$, and $s$ denotes the sign bit of the SM stochastic stream, such that $s = 0$ for $m \geq 0$ and $s = 1$ otherwise.

Let $c$ be the product of the continuous values $b$ and $m$ ($c = b \times m$). In the stochastic domain, this multiplication is performed through element-wise multiplication between the bipolar stream $\mathbf{B}$ and the SM stream $\mathbf{M}$, such that:

$$c_t = f(b_t, m_t, s), \tag{3.8}$$

where $c_t$ is an element of the stochastic stream $\mathbf{C}$ and function $f(\cdot)$ executes logical binary operations, ensuring:

$$\begin{aligned} \mathbb{E}[c_t] &= c \\ &= b \times m. \end{aligned} \tag{3.9}$$

By combining Equation (3.6), Equation (3.7), and Equation (3.9), the following is obtained:

$$\mathbb{E}[c_t] = \mathbb{E}[(-1)^{1-b_t}] \cdot \mathbb{E}[(-1)^s \cdot m_t], \tag{3.10}$$

Assuming that $b_t$ and $m_t$ are independent, Equation (3.10) can be rewritten as:

$$\begin{aligned} \mathbb{E}[c_t] &= \mathbb{E}[(-1)^{1-b_t} \cdot (-1)^s \cdot m_t], \\ &= \mathbb{E}[(-1)^{1-b_t+s} \cdot m_t], \\ &= \mathbb{E}[(-1)^{s_t} \cdot m_t], \end{aligned} \tag{3.11}$$

where $s_t \in \{0, 1\}$ and is logically equivalent to $b_t \odot s$ ($b_t$ XNOR $s$), as shown in Table 3.1.

According to Equation (3.11), $c_t$ is a ternary variable, i.e., $c_t \in \{-1, 0, +1\}$, and can be encoded using two binary bits $c_t(0)$ and $c_t(1)$ such that:

$$c_t = (-1)^{c_t(1)} \cdot c_t(0), \tag{3.12}$$

where $c_t(0) = m_t$ and $c_t(1) = s_t$. Therefore, each element of the stochastic stream $\mathbf{C}$ can be represented with a sign bit $c_t(1) = b_t \odot s$ and a magnitude bit $c_t(0) = m_t$. In other words, the multiplication of SM and bipolar streams produces an SM stream in which each magnitude bit is associated with a corresponding individual sign bit. We designate this new type of stochastic stream as the Dynamic Sign-Magnitude (DSM) stream. In DSM streams, positive and negative elements are represented with $s_t = 0$ and $s_t = 1$, respectively. Consequently, ternary values $\{01\}$, $\{11\}$, and $\{00, 10\}$ are interpreted as representing +1', -1', and '0', respectively.

To illustrate the multiplication process between a bipolar and an SM stochastic stream, consider $\mathbf{B} = \{1, 1, 1, 0, 1, 1, 1, 0\}$ and $\mathbf{M} = (s = 1)\{0, 0, 0, 0, 1, 1, 1, 1\}$ representing a bipolar stream with an expected value of $\mathbb{E}[b_t] = \frac{1}{8}(1 + 1 + 1 - 1 + 1 + 1 + 1 - 1) = 0.5$ and an SM stream with the expected value of $\mathbb{E}[(-1)^s \cdot m_t] = \frac{-1}{8}(0 + 0 + 0 + 0 + 1 + 1 + 1 + 1) = -0.5$. The resulting DSM stream $\mathbf{C} = \{10, 10, 10, 00, 11, 11, 11, 01\}$ has an expected value of $\mathbb{E}[c_t] = \frac{1}{8}(0 + 0 + 0 + 0 - 1 - 1 - 1 + 1) = \frac{-2}{8} = -0.25$, which aligns with the expected multiplication result.

The advantages of utilizing the SM representation for weights during forward propagation are outlined as follows. First, SM representation enables more precise multiplications compared to bipolar stochastic streams. Additionally, combining SM and bipolar representations allows for efficient multiplication operations using only a single XNOR gate per stream. The resulting multiplications, represented in the DSM format, can be seamlessly integrated to produce integral streams, which serve as inputs to the *IStanh* non-linear activation function. In Section 3.5, we will demonstrate the impact of using the SM representation for weights on the accuracy performance of our SC-based neural network, compared to the conventional

Table 3.1: Truth table for calculating the dynamic sign bit of DSM stochastic streams.

| $b_t$ (positive/negative) | $s$ (positive/negative) | $s_t$ (positive/negative) |
|---|---|---|
| 0 (-) | 0 (+) | 1 (-) |
| 0 (-) | 1 (-) | 0 (+) |
| 1 (+) | 0 (+) | 0 (+) |
| 1 (+) | 1 (-) | 1 (-) |

bipolar representation.

### 3.4.3 Forward/Backward Propagation

The forward propagation of the proposed SC-based neural network is carried out in two main steps, as described in Algorithm 5:

**Step-1**: A stochastic sampling operation is performed on all inputs (in bipolar format) and weights (in SM format). The sampled inputs and weights are denoted as $x^b$ and $w^b$, respectively. This step is represented in Algorithm 5 using the functions $StochasticSample\_B()$ and $StochasticSample\_SM()$.

**Step-2**: In the first layer of the SC-based NN, the samples $x^b$ and $w^b$ are multiplied, and their sum of products is computed. This sum forms an element in an integral stochastic stream, which is subsequently passed through the *IStanh* activation function. The bipolar output of *IStanh*, denoted as $h^b$, is then multiplied with $w^b$ in the second layer, and their sum of products is passed to the next activation function. This process is repeated layer by layer until the final layer is reached. It is important to note that the output layer does not include a non-linear activation function; therefore, only the sum of products, denoted as $y$, is computed.

The forward propagation process is repeated $L$ times, where $L$ represents the length of the stochastic sequence. The summations of $h^b$ and $y$ across these iterations are denoted as $H$ and $Y_o$, respectively. For backpropagation, real-valued node activations are required. These real values are obtained by calculating the average of the activation function outputs, dividing $H$ by $L$. Similarly, the output $Y_o$ is divided by $L$ to compute the real-valued output, which is used in the loss function. If we set the length $L$ of stochastic streams to a power of two number, these divisions can be implemented efficiently using bit-shift operations. Finally, the classification result, $Y_p$, is determined using the *argmax* function.

It is worth emphasizing that the input data $x^b$ and weights $w^b$ are represented in bipolar and SM formats, respectively. The outputs of hidden layers ($h^b$) are represented in bipolar format, while the final outputs ($y$) are represented in integral format, as the output layer lacks a non-linear activation function. An illustration of the forward propagation process for the SC-based NN is provided in Figure 3.3.

We minimize the loss for each output node individually using the Hinge Loss function, which is defined as:

$$HingeLoss = max(0, 1 - y_o \cdot y_t), \tag{3.13}$$

where $y_t \in \{-1, +1\}$ represents the ground truth label of the input data and $y_o \in [-1, 1]$

**Algorithm 5:** Pseudo code of the proposed algorithm for forward propagation, where $L$ is the length of stochastic sequence, $K$ is the number of layers including output layer, and $StochasticSample\_SM$ and $StochasticSample\_B$ are the functions that output stochastic samples in SM and bipolar formats, respectively. 

**Data:** Input data $X \in [-1, +1]$ and weights $W \in [-1, +1]$

**Result:** Output $y_o$, activation derivatives with respect to its input $h'$, activation values H and predicted class $y_p$.

**for** $j = 1 : L$ **do**
  **for** $i = 1 : K$ **do**
    $w_i^b \leftarrow StochasticSample\_SM(W_i)$
    **if** $i == 1$ **then**
      $x^b \leftarrow StochasticSample\_B(X)$
      $h_i^b \leftarrow IStanh(\sum x_i^b \; w_i^b)$
    **else if** $i == K$ **then**
      $y \leftarrow \sum h_{K-1}^b \; w_K^b$
    **else**
      $h_i^b \leftarrow IStanh(\sum h_{i-1}^b \; w_i^b)$
    **end**
  **end**
  $H \leftarrow H + h^b$
  $Y_o \leftarrow Y_o + y$
**end**
$H \leftarrow H/L$
**if** $|H| < 1$ **then**
  $h' \leftarrow 1$
**else**
  $h' \leftarrow 0$
**end**
$y_o \leftarrow Y_o/L$ and $y_p \leftarrow argmax(Y_o)$

**Algorithm 6:** Pseudo code of the proposed algorithm for back-propagation and updating weights. * denotes matrix multiplication. ©IEEE 2021 [1].

**Data:** Input data $X \in [-1, +1]$, current weights $W^t \in [-1, +1]$, neurons values $H$, learning rate $\alpha$ , loss derivative $loss'$ and activation derivatives $\partial h$

**Result:** Updated Weights $W^{t+1}$

$w^b \leftarrow sign(W^t)$ and $H \leftarrow sign(H)$
**for** $i = K : 1$ **do**
  **if** $i == 1$ **then**
    $\nabla w_1 \leftarrow sign(X * grad_1)$
  **else if** $i == K$ **then**
    $grad_{L-1} \leftarrow sign(\sum loss' * w_L^b) h'_{L-1}$
    $\nabla w_L \leftarrow sign(H_{L-1} * loss')$
  **else**
    $grad_{i-1} \leftarrow sign(\sum grad_i * w_i^b) h'_{i-1}$
    $\nabla w_i \leftarrow sign(H_{i-1} * grad_i)$
  **end**
**end**
$W^{t+1} \leftarrow W^t - \alpha \nabla w$

denotes the predicted output obtained from Algorithm 5. The Hinge Loss penalizes predictions that do not meet the margin of correctness, 1, by encouraging the model to produce a prediction $y_o$ that has the same sign as the target $y_t$ and is at least 1 unit away from the decision boundary. When $y_o \cdot y_t \geq 1$, the loss becomes zero, indicating a correct prediction with sufficient confidence.

The partial derivative of the Hinge Loss with respect to the output $y_o$ can be simply calculated as:

$$\frac{\partial Loss}{\partial y_o} = loss' = \begin{cases} -y_t & \text{if } y_o y_t < 1 \\ 0 & \text{otherwise} \end{cases}. \tag{3.14}$$

It is important to note that the minimization of the Hinge Loss function does not involve any real-valued (i.e., full-precision) multiplications, as the derivative of the loss, $loss'$, takes discrete values in $\{-1, 0, 1\}$.

In SC-based neural networks, two primary challenges arise when performing backpropagation:

- **Loss of Gradient Precision:** The precision of gradients decreases significantly during backpropagation, which is considered critical for the successful training of models using gradient descent algorithms [48].

- **Inaccurate Derivative Computation**: It is inherently challenging to accurately compute the derivative of stochastic activation functions.

To address these challenges, we propose the following solutions:

- **Gradient Polarity Estimation:** Instead of calculating precise gradient values during backward propagation, we estimate the polarity of the gradients. Using this estimated polarity, we update the weights with a small learning rate. This approach simplifies the backpropagation process while maintaining the directionality of gradient updates.

- **Straight-Through Estimator:** To pass the gradients through neurons, we use "straight-through estimator" method, as suggested by Hubara et al. [56], which performs $h' = 1_{|H|<1}$, where $h'$ represents the derivative of the activation function with respect to its input. The estimator effectively blocks gradient flow through saturated nodes, which are nodes whose absolute value equals or exceeds 1. Notably, in our SC-based NN, the absolute value of nodes is always constrained to be less than or equal to 1, as the activation functions output bipolar values. This ensures compatibility with the straight-through estimator method.

The implementation details of this procedure are outlined in Algorithm 5.

To estimate the polarity of gradients, we binarize both the weights and nodes deterministically, using the sign function, with the exception of the nodes in the first layer. The weights in the first layer are updated using the real values of the input nodes, $X$. Following the loss function equation in Equation (3.14) and applying the "straight-through estimator," we express the gradients as follows:

$$g_j^{(\ell)} = \begin{cases} -y_{t_{(y_o y_t<1)}} & \text{output layer} \\ 1_{|f(v_j^{(\ell)})|<1} sign(\sum_m g_j^{(\ell+1)} sign(w_{m,j}^{(\ell)})) & \text{middle layers} \end{cases}, \tag{3.15}$$

and

$$\nabla w_{j,k}^{(\ell)} = g_j^{(\ell)}\ sign(f(v_j^{(\ell-1)})), \tag{3.16}$$

where all multiplications are performed using ternary operations. Unlike the forward propagation, during backpropagation, both the weights and nodes are deterministically binarized using the sign function and represented using a sequence length of one.

To update the weights, the learning rate in Eq. (3.4) is treated as a power-of-two number, facilitating its implementation via shift operations. The learning rate determines the required

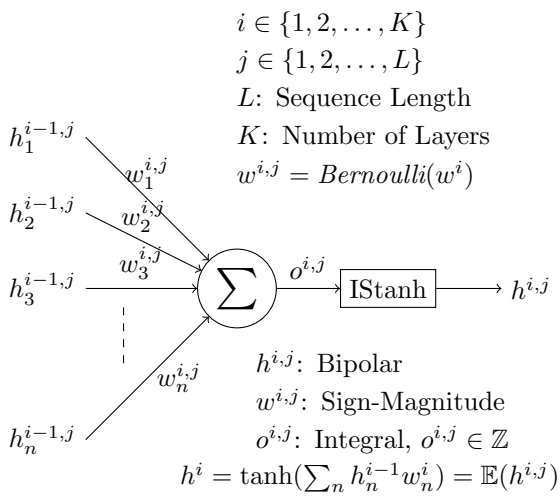



Figure 3.3: Forward propagation. ©IEEE 2021 [1].

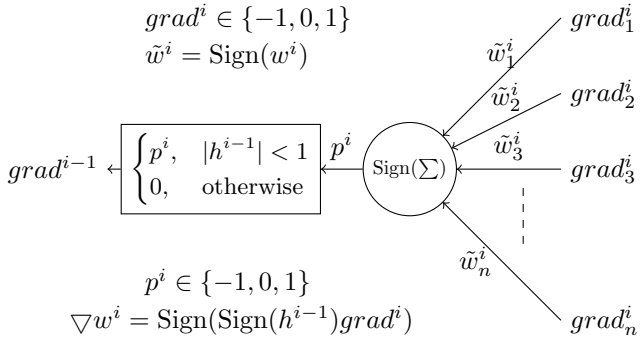



Figure 3.4: Backward propagation. ©IEEE 2021 [1].

precision for storing real values of the weights in memory. For instance, when the learning rate is $\frac{1}{2^N}$, only $N + 1$ bits are necessary for the weights (1 sign bit and $N$ fractional bits). The backward propagation procedure is detailed in Algorithm 6 and illustrated in Figure 3.4.

## 3.5 Experimental Results

For evaluation purposes, we use MNIST image classification benchmark. MNIST dataset contains 60K and 10K of training and test samples, respectively. In all experimental results, our SC-based NNs are trained with all the training samples and we report the accuracy obtained from the test set. In all the experiments in this paper, we use 16-bit fixed-point multipliers to perform multiplications of full-precision models. Moreover, the real values of weights and inputs are represented using 16-bit and 8-bit fixed-point formats in our SC-based NNs, respectively. Finally, the sequence lengths provided in the results tables indicate the

Table 3.2: Accuracy of our SC-based NN with 784-128-128-10 Fully-Connected Network Configuration on MNIST dataset. ©IEEE 2021 [1].

| Model | sequence length | weight binarization | Accuracy (%) |
|---|---|---|---|
| Full-Precision | N/A | N/A | 98.30 |
| SC-based NN | 1 | SM | 51.04 |
| SC-based NN | 2 | SM | 95.54 |
| SC-based NN | 4 | SM | 96.66 |
| SC-based NN | 8 | SM | 97.23 |
| SC-based NN | 16 | SM | 97.52 |
| SC-based NN | 32 | SM | 97.58 |
| SC-based NN | 64 | SM | 97.61 |
| SC-based NN | 128 | SM | 97.66 |
| SC-based NN | 256 | SM | 97.65 |
| SC-based NN | 512 | SM | 97.66 |
| SC-based NN | 16 | Bipolar | 96.40 |

sequence length of stochastic streams during inference (or forward propagation) and the sequence length of the binarized parameters for the backward propagation during the training of the networks is always one.

### 3.5.1 Shallow FCNNs

Table 3.2 summarizes the classification accuracy of our FCNN with two hidden layers of size 128 (i.e., the network configuration of 784-128-128-10) trained using the conventional GD algorithm and our proposed training method. It is worth noting that the conventional GD algorithm performs both the inference and training computations in full-precision whereas our proposed training method only uses binary/ternary operations for both the inference and training processes. We use different stochastic sequence lengths ranging from 1 to 512 when using our training method. The batch size is set to 100 and the networks are trained for 500 epochs. The simulation results show that the classification accuracy of our SC-based NN improves as the sequence length increases and the stochastic computations become more accurate. However, the accuracy plateaus for the sequence lengths of 256 and 512 as our SC-based NN reaches its full learning capacity. Furthermore, Table 3.2 shows the accuracy of a SC-based NN trained with a sequence length of 16 with the bipolar representation of the weights. As discussed in 3.4.2, the near-zero value of weights causes accuracy degradation in stochastic computations where streams are represented in bipolar format. Our results show accuracy degradation of 1.12% between bipolar and SM representations of the weights.

Table 3.3: Accuracy of our SC-based NN with 784-1024-1024-1024-10 Fully-Connected Network Configuration on MNIST dataset. ©IEEE 2021 [1].

| Model | Sequence Length | Employed technique | Accuracy (%) |
|---|---|---|---|
| Full-Precision | N/A | - | 98.4 |
| Full Precision | N/A | Dropout | 98.65 |
| SC-based NN | 16 | - | 98.03 |
| SC-based NN | 16 | Dropout | 98.23 |
| SC-based NN | 16 | Stochastic binarization during back-propagation | 98.12 |
| SC-based NN | 256 | - | 98.11 |

### 3.5.2 Deep FCNNs

To ensure that our training method works properly on deeper networks, we train a FCNN with three hidden layers of size 1024 (i.e., the network configuration of 784-1024-1024-1024-10) on MNIST dataset. We also employ the dropout technique for our implementations and show that it improves upon our training method (see Table 3.3). This is due to the fact that the dropout technique improves regularization of NNs by preventing them from overfitting the training set. In Algorithm 6, we described how to back-propagate using our gradient estimation method and deterministic binarization of weights and nodes. When using deterministic binarization to estimate the gradients, small (e.g., 0.0001) and large (e.g., 0.9) values of weights and nodes will have the same impact. To better estimate the gradients, we employ stochastic binarization during back-propagation. More precisely, we stochastically take one sample from weights and nodes when calculating the gradients during the back-propagation using SM representation. In other words, the real value of the weights and nodes are converted to ternary values represented with SM streams that only have a sequence length of one. This will also resolve the issue with the first layer as we stochastically sample from the value of input nodes. Note that the sequence length for back-propagation is always one, meaning that only one sample is taken during back-propagation. Table 3.3 reports the test accuracy of our proposed SC-based NNs when using stochastic binarization during back-propagation. It is evident that performing back-propagation using stochastic binarization to compute and back-propagate the gradients produces higher accuracy compared to the one that performs back-propagation with deterministic binarizated parameters. Furthermore, we employed the dropout technique in our SC-based NN which improves the accuracy performance by 0.2%.

### 3.5.3 Comparison with Existing Works

Table 3.4 provides the accuracy performance of our proposed SC-based NNs and other existing binarized neural networks on MNIST dataset. It should be noted that none of the networks in Table 3.4 are entirely binarized except the FCNN proposed by Liu et al. [17] and ours. Other works either proposed binarization solely for the inference process or have trained their binarized networks with full-precision local gradients. In addition to FCNNs, we show that our approach is also compatible with convolutional networks by training a simple CNN on MNIST dataset using Algorithm 5 and Algorithm 6. The under-test CNN consists of two convolutional layers with the filter size of $3 \times 3$ followed by a fully-connect layer with 128 neurons. The number of filters of the first two convolution layers are 32 and 64 (i.e., the network configuration of 784-32(3)-64(3)-10), respectively. Table 3.4 provides the accuracy results of the existing works from their proposed SC-based NNs with shortest reported sequence length. The experimental results show that our SC-based NNs outperform all the SC-based models in terms of processing latency (i.e., sequence length) by up to 128× while maintaining comparable accuracy. Moreover, our proposed training method replaces all binary-radix multipliers with ternary operations, enabling low-cost on-chip learning. For instance, a single full-precision multiplier (i.e., 16-bit multiplier) requires 3116 $\mu m^2$ in TSMC 65-$nm$ CMOS technology while a ternary multiplier requires 269× less silicon area.

## 3.6 Discussion

In [17], extended stochastic logic is employed to implement both forward propagation and backward propagation in a multilayer perceptron. By leveraging a binary search mechanism, a reconfigurable stochastic computational activation unit, and an LFSR sharing scheme, the design achieves reduced area and energy consumption compared to binarized neural networks and traditional floating-point and fixed-point implementations. However, the use of extended stochastic logic in [17] necessitates an additional stochastic divider and extra computation time to convert stochastic sequences back into binary representations. In contrast, our proposed design eliminates the need for such conversions, as the calculated gradients are inherently ternary. Moreover, the method in [17] requires relatively long sequence lengths to achieve high accuracy, leading to significant latency (256 clock cycles with 16× parallelization). In contrast, the method proposed in this chapter achieves comparable accuracy to that reported in [17], using a sequence length of only 16 during forward propagation and a sequence length of 1 during backward propagation, while maintaining a similar network size and structure.

Table 3.4: Accuracy Comparison of our Proposed and Existing Binarized Neural Networks on MNIST dataset. ©IEEE 2021 [1].

| Model | Type | Hidden Layer Configuration | Sequence Length | Accuracy (%) |
|---|---|---|---|---|
| This Work | Fully Connected | 128-128 | 16 | 97.52 |
| This Work | Fully Connected | 1024-1024-1024 | 16 | 98.28 |
| TCADICS'18 [48] | Fully Connected | 128-128 | N/A[a] | ≈97 |
| TVLSI'17 [49] | Fully Connected | 300-600 | 1024 | 97.99 |
| DAC'16 [57] | Fully Connected | 100-200 | 1024 | 97.59 |
| TCOMP'18 [17] | Fully Connected | 200-100 | 256 | 97.95 |
| This Work | Convolutional | 32(3)-64(3)-128 | 16 | 98.38 |
| DAC'18 [55] | Convolutional | LeNet-caffe[b] | 32 | ≈98.8 |
| ASP-DAC'17 [54] | Convolutional | LeNet-5[b] | 64 | 95.6 |
| ICRC'16 [58] | Convolutional | LeNet-5 | 128 | 86.12 |
| SIGOPS'17 [53] | Convolutional | LeNet-5 | 256 | 98.26 |

[a] SC is only used when updating the weights

[b] Note that our CNN model, LeNet-5 and LeNet-caffe have around 76000. 81000 and 656000 trainable parameters, respectively.

In [59], gradient compression is employed to reduce communication overhead during distributed training. Gradients are stochastically compressed to three discrete levels, {-1, 0, +1}, significantly lowering communication costs. Similarly, [48] adopts a ternary gradient compression strategy to minimize computational overhead when updating weights. However, in both methods, backpropagation computations are still performed in real-valued (binary-radix) formats. Unlike these approaches, our method executes all backpropagation computations entirely using ternary operations, providing a novel and efficient solution for gradient computation and weight updates.

## 3.7 Conclusion

SC-based NNs were initially proposed as a low-cost alternative for hardware implementations of neural networks. While SC-based implementations offer a significantly smaller hardware footprint compared to their binary-radix counterparts, they typically require long stochastic sequence lengths (often exceeding 256) to mitigate the accuracy loss inherent to SC. Consequently, SC-based NNs face challenges related to high computational latency and increased energy consumption, making them less efficient than binary-radix implementations.

In this work, we propose a novel approach capable of performing both inference and training processes using significantly shorter stochastic sequence lengths (e.g., a sequence length of 16), thereby addressing these limitations.

Moreover, we explored the possibility of training neural network with ternary gradients where gradients are computed using ternary operations. We proposed a training algorithm to train neural networks using SC on MNIST dataset. Our experimental results promise the possibility of training neural networks, more specifically, binarized neural networks (e.g., SC-based neural networks), with binary/ternary operation. We also proposed a new stochastic representation (i.e., DSM) used in our SC-based neural networks which enabled us to perform both the inference and training processes on a small sequence length (i.e., sequence length of 16). As future work, we plan to apply our training method on more challenging datasets such as CIFAR10 and ImageNet.

# 4

# Shift-based Neural Network

Deep neural networks (DNNs) have demonstrated remarkable performance across a wide range of applications, including image classification and natural language processing. However, deploying state-of-the-art (SOTA) DNNs requires high computational resources and specialized hardware, such as GPUs, due to the extensive use of costly high-precision multiplications [15]. Additionally, modern DNNs are often over-parameterized, leading to substantial memory usage and increased data movement between computation and memory units, which further exacerbates energy and latency constraints [60]. These challenges make it particularly difficult to implement DNNs on resource-limited hardware platforms, such as mobile devices and embedded systems. To address these challenges, significant research has been conducted to reduce the computational and memory costs of DNN deployment while maintaining model accuracy. Approaches include efficient architecture designs [6–9], network pruning [61, 62], stochastic computing [16, 63], and spiking neural networks [64, 65], among others.

Another prominent direction focuses on the quantization of DNNs, where continuous real-valued weights and activations are mapped to discrete integer values [66]. Quantization significantly reduces computational complexity and memory requirements, making DNNs more suitable for low-power and resource-constrained environments. While uniform quantization methods —where discrete levels are evenly spaced— have been widely studied [67–69], recent

research has explored non-uniform quantization techniques, which allow discrete levels to vary in step size for improved efficiency and performance [11, 70–72].

We focus on advancing the quantization of DNNs by exploring both uniform and non-uniform quantization strategies. The primary objective of this study, as detailed in this chapter, is to develop a non-uniform quantization framework that quantizes DNN weights into discrete power-of-two values (i.e., $\pm 2^n$). This approach aims to achieve accuracy equivalent to full-precision models while utilizing the minimum possible number of bits. By leveraging this framework, computationally expensive full-precision multiplications can be replaced with efficient shift-add operations, thereby addressing the significant computational challenges associated with deploying neural networks on resource-constrained platforms.

## 4.1 Related Work

Quantization methods can generally be classified into three main categories. The first category focuses on minimizing quantization error, where real-valued weights and activations are mapped to discrete levels that accurately represent the original data [70, 73–76]. Recent methods in this category primarily aim to achieve a precise representation of data by minimizing the discrepancy between the quantizer's input and output values. This process can be performed either *offline*, using structural information derived from a pre-trained full-precision network, or *online*, where quantization is optimized dynamically during training. For instance, in the Half-wave Gaussian Quantization (HWGQ) scheme [74], the quantization error is minimized using Lloyd's algorithm [77]. This method leverages the statistical distribution of activations obtained from the full-precision network to fit the quantizer to the data. However, a key limitation of HWGQ lies in its offline optimization, where the activation distributions of the full-precision network may differ from those of the quantized network, potentially compromising performance. To address this issue, the learned linear symmetric quantizer (LLSQ) method employs an online approach, optimizing quantization error during the training phase to better adapt to dynamic network parameters [76]. Similarly, LQ-Net introduces a non-uniform quantization scheme that reformulates the quantization error minimization problem as a linear regression task with a closed-form solution [70]. In the TSQ method, quantization error is minimized by solving a non-linear least square regression problem [75]. In the TW-networks, weights are quantized into ternary values $\{+1, 0, -1\}$ by minimizing the Euclidian distance between the full precision and the ternary-valued weights [73]. While these methods provide rigorous quantization error minimization strategies, they have been empirically outperformed by approaches in the second category, suggesting that minimizing

quantization error alone may not be the most effective way to achieve high-performance quantized networks. This observation highlights the need for alternative strategies that prioritize the overall accuracy performance of the network.

In the second category, quantization is formulated as an optimization problem that is jointly optimized with the loss function during the training phase. In this approach, quantizers are redesigned with learnable parameters, enabling the identification of optimal quantization intervals. Unlike the first category, where the primary objective is to minimize quantization error for accurate data representation, the second category focuses on optimizing the quantization process to achieve the best possible accuracy performance. Notable methods in this category include PACT [67], QIL [69], and LSQ [68], which demonstrate the efficacy of integrating quantization optimization into the overall training process.

Finally, the third category focuses primarily on the training strategies for quantized networks, aiming to enhance their performance accuracy [72, 78]. This includes techniques and methodologies designed to improve the effectiveness of quantized networks during the training process. For example, progressive training frameworks, combined with knowledge distillation, have been shown to significantly enhance the performance of the DoReFa-Net quantization method across various tasks [66, 78]. These approaches highlight the importance of tailored training techniques in addressing the accuracy challenges associated with quantized networks.

### 4.1.1 Motivation

In this work, we aim to bridge the gap between existing quantization methods by leveraging the statistical characterization of the weight and activation distributions during the quantization process. Specifically, we propose a novel quantization scheme that utilizes the standard deviation of weight and activation distributions during the training phase to identify the optimal quantizer, which maximizes accuracy performance using task loss and back-propagation. The contribution of standard deviation in enhancing the proposed quantization method is detailed in Section 4.3. Furthermore, in Section 4.4, we introduce two training techniques designed to further improve the performance of quantized networks.

Unlike prior methods that predominantly focus on either uniform or non-uniform quantization, our proposed framework is versatile and supports both approaches. For non-uniform quantization, we employ power-of-two discrete levels and demonstrate that this method surpasses state-of-the-art results [11] while achieving a 10× faster convergence rate.

Quantizers with discrete intervals that include a zero level inherently benefit from pruning,

as values mapped to this level during quantization are set to zero and can be pruned. This property enhances the desirability of quantization methods compared to other cost-reduction techniques. However, only a limited number of studies have explored the interplay between pruning and quantization or their mutual impact [61,69]. In this work, we not only investigate the pruning property of quantization in greater depth but also demonstrate that our proposed method enables flexible adjustment of the pruning ratio during the quantization process. For instance, when performing an image classification task on the ImageNet dataset, we demonstrate that up to 40% of the weights in a 3-bit ResNet-18 model can be pruned while incurring less than a 1% accuracy loss compared to its full-precision counterpart.

## 4.2 Standard Deviation-based Quantization

To establish a connection between existing quantization methods, we address the following question: *"How much of the information is important in a quantized network?"* To answer this, a closer examination of the statistical characterization of weights and activations is necessary. Prior research has demonstrated that the outputs of convolutional and fully connected layers often exhibit a bell-shaped distribution, particularly when followed by batch-normalization layers [79]. Similarly, weights regularized using the L2 regularization method tend to follow a bell-shaped distribution as well. The width of this bell-shaped distribution is determined by its standard deviation, $\sigma$. For instance, in a Gaussian distribution, approximately 99.73% of the data points lie within three standard deviations from the mean $\mu$. However, not all these values contribute equally to the overall accuracy and performance of a neural network. To address this issue, we propose a standard deviation-based clipping function with a learnable parameter $\alpha$, capable of identifying the values that are most significant to the network. The foundation of our method builds upon the PACT framework [67]. We extend PACT by integrating the standard deviation of weights and activations into the clipping function as follows:

$$\mathrm{y}(x) = \begin{cases} x & |x| < \alpha\sigma \\ \mathrm{sign}(x) \cdot \alpha\sigma & |x| \geq \alpha\sigma \end{cases}, \tag{4.1}$$

where $\alpha$ is a learnable parameter of the quantizer, and $\sigma$ is the standard deviation of the weight or activation distribution. For activations, we further integrate the clipping function

with the ReLU activation function to eliminate negative values, resulting in:

$$\mathrm{y}(x) = \begin{cases} 0 & x \leq 0 \\ x & 0 < x < \alpha\sigma \\ \alpha\sigma & x \geq \alpha\sigma \end{cases}. \tag{4.2}$$

The proposed function effectively discards outlier values that are further than $\alpha\sigma$ from the mean of the weight or activation distribution. Notably, for weights, we assume the mean to be zero throughout, as experimental observations indicated that weight distributions had a near-zero mean, and omitting it did not impact performance. For activations, the mean is inherently non-zero; therefore, to ensure the standard deviation is calculated correctly, we preprocess activations by removing all negative values, mirroring the remaining positive values horizontally, and recalculating the distribution. This ensures that the mean of the activation distribution becomes zero prior to applying the clipping function. By utilizing this clipping strategy, we improve the quantization process by dynamically controlling the contribution of values based on their statistical significance.

To address the variability in the standard deviation ($\sigma$) of activations across different data batches, we adopt an approach inspired by batch-normalization. Unlike the standard deviation of weights, which remains relatively constant, the standard deviation of activations can vary significantly with each new batch of data. To stabilize this variation during training, we compute a running average of $\sigma$ using the moving average with a momentum factor of 0.001, as follows:

$$\hat{\sigma}_{\text{new}} = (1 - \text{momentum}) \times \hat{\sigma} + \text{momentum} \times \sigma_t, \tag{4.3}$$

where $\hat{\sigma}$ represents the running average, and $\sigma_t$ is the standard deviation of the current batch. The momentum value of 0.001 was determined empirically through experiments, ensuring stability and optimal results. Once the clipping function output is obtained, the values are quantized into $L_P + L_N + 1$ discrete integer levels using $b$ bits. This is expressed as:

$$y_d = clip(\left\lfloor y \cdot \frac{L_P}{\alpha\sigma} \right\rceil, -L_N, L_P), \tag{4.4}$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer, and $y_d \in \mathbb{N}$. The values of $L_P$ and $L_N$ are determined by the bit-width $b$ and the data type (signed or unsigned). For unsigned data (e.g., activations), $L_N = 0$ and $L_P = 2^b - 1$, while for signed data (e.g., weights), $L_P = L_N = 2^{b-1} - 1$. Notably, our method symmetrically quantizes signed values; for

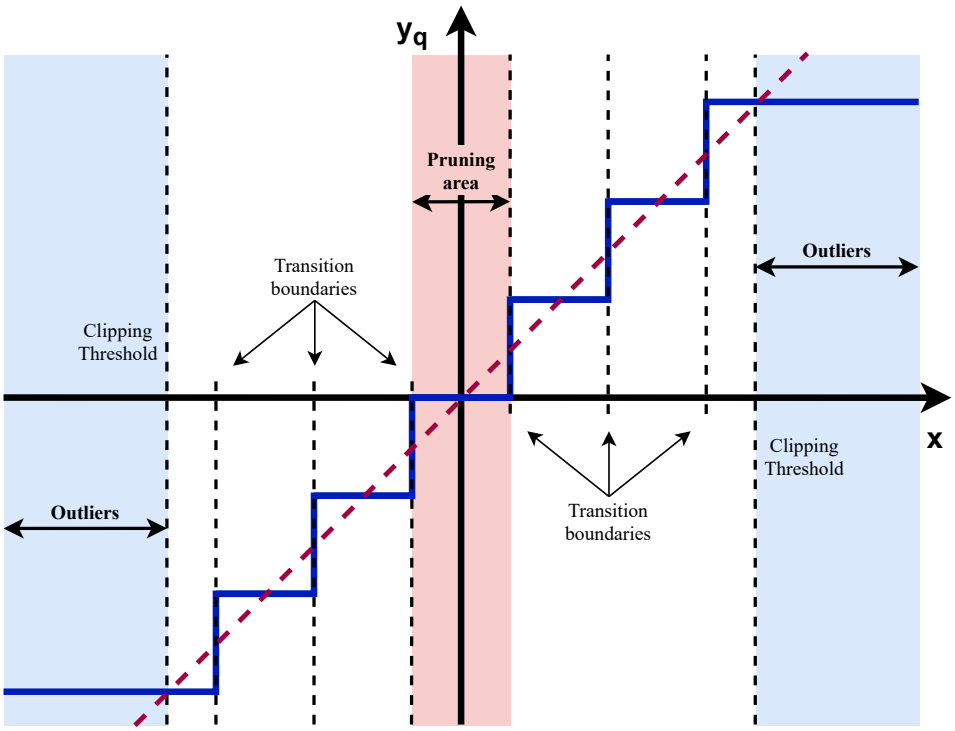



Figure 4.1: An example of a 3-bit quantizer for signed values.

example, a 2-bit signed quantization results in three discrete levels (i.e., ternary values). The final step in the quantization process involves scaling the quantized values, $y_d$, to obtain the quantized output, $y_q$, as follows:

$$y_q = y_d \cdot \frac{\alpha\sigma}{L_P}, \tag{4.5}$$

where $y_q \in \mathbb{R}$. This scaling operation can be seamlessly integrated into the batch-normalization layer, eliminating any additional computational overhead.

Our proposed quantization method inherently supports pruning by setting values within a predefined pruning area to zero during the quantization process. Specifically, any value satisfying the condition

$$\text{if} \quad |y| < \frac{\alpha\sigma}{2L_P}, \quad \text{then} \quad y_d = 0, \tag{4.6}$$

is pruned, as these values fall below the quantization threshold. Consequently, the pruning ratio is directly influenced by the clipping threshold $\alpha\sigma$, with higher thresholds leading to higher pruning ratios. An illustrative example of a 3-bit quantizer for signed values is shown in Figure 4.1, demonstrating the relationship between quantization and pruning.

### 4.2.1 Optimizing $\alpha$ with Backpropagation

Following the principles introduced in the PACT method and utilizing the Straight-Through Estimator (STE) [80], we derive the gradient for the quantizer parameter $\alpha$ as follows:

$$g_\alpha = \begin{cases} \sigma \cdot g_y & x \geq \alpha\sigma \\ 0 & \text{otherwise} \end{cases}, \tag{4.7}$$

where $g_y$ represents the incoming gradient from the subsequent layers. This gradient expression ensures that the quantization operation, which involves hard thresholds, can be differentiated during the backpropagation step by treating it as a straight-through operation. Similarly, the gradient with respect to the activation input, $g_x$, is computed using the following expression:

$$g_x = \begin{cases} g_y & 0 < x < \alpha\sigma \\ 0 & \text{otherwise} \end{cases}. \tag{4.8}$$

Here, $g_x$ represents the gradient for the input to the quantizer, where the gradient is only propagated through the quantizer for values of $x$ that lie within the quantization range $(0 < x < \alpha\sigma)$.

To further stabilize the training process and prevent the quantizer parameter $\alpha$ from exploding or vanishing, we introduce a gradient scale $s$ and a weight decay term $\lambda\alpha$ to the gradient of $\alpha$. This results in the modified gradient expression:

$$g_\alpha = \begin{cases} s\sigma \cdot g_y + \lambda\alpha & x \geq \alpha\sigma \\ 0 & \text{otherwise} \end{cases}. \tag{4.9}$$

In this revised formulation, the gradient scale $s$ helps control the magnitude of the gradient, while the weight decay term $\lambda\alpha$ encourages regularization of the quantizer parameter $\alpha$, thereby controlling the pruning ratio. This addition ensures that $\alpha$ does not grow excessively large, and helps guide the quantizer's behavior during the optimization process.

### 4.2.2 Non-Uniform Power-of-two Quantization

To quantize weights to discrete values that are powers of two, we can leverage the clipping function described in Equation (4.1), which helps map the weights to the set $0, \pm 2^k$, where $k \in \mathbb{Z}_0^+$. This non-uniform quantization approach allows us to replace the computationally expensive multiplications with simple shift and addition/subtraction operations, which significantly reduces the computational cost. To achieve this power-of-two quantization, we

modify Equation (4.4) to generate discrete integer values $y_{int} \in \mathbb{N}$ as follows:

$$y_{int} = \left\lfloor \log_2(|y| \cdot \frac{L_{p2}}{\alpha\sigma}) \right\rceil, \tag{4.10}$$

where $L_{p2} = 2^{2^{b-1}-2}$ represents the range of the power-of-two quantized values. The integer value $y_{int}$ is then transformed into power-of-two discrete values, including zero, using the following formula:

$$y_{p2} = \begin{cases} clip(\text{sign}(y) \cdot 2^{y_{int}}, -L_{p2}, L_{p2}) & y_{int} \geq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{4.11}$$

For example, when using 3-bit quantization ($b = 3$), the weights can be quantized into seven discrete values, i.e., $y_{p2}^{3-bit} \in 0, \pm 1, \pm 2, \pm 4$. This approach ensures that the quantized weights are restricted to powers of two, facilitating efficient hardware implementations. The quantization process is then completed by applying the quantizer scale to $y_{p2}$, as described in Equation (4.5). Finally, the quantizer parameter $\alpha$ is optimized using the same gradient-based approach outlined in Equation (4.9), which allows for fine-tuning of the quantization process during training.

## 4.3 Contributing Factors

The main intuition behind our heuristic approach for parameterizing quantizers using the statistical structure of the weights and activations is to incorporate the standard deviation of the data distribution during the quantization process. This approach contrasts with previous parameterization methods by utilizing not only the input samples but also the statistical characteristics (specifically, the standard deviation) of weights and activations from each layer in the network. The standard deviation, in particular, indicates the density of a distribution. For example, in a distribution with a small standard deviation (i.e., a highly dense distribution), where most of the data resides within the pruning area, it is critical to prevent outliers from influencing the clipping threshold significantly. In such cases, the small value of $\sigma$ ensures that the gradients from the outliers do not substantially shift the clipping threshold, as described in Equation (4.9). In addition to this explanation, we identify three additional contributing factors that enhance the effectiveness of our method: (1) inclusivity, (2) adaptive gradient scale factor, and (3) faster convergence.

**Inclusivity**: Unlike PACT, where the gradients of the quantizer parameter only rely on

the outliers, the gradients in our method are sensitive to the entire dataset. Specifically, we introduce the standard deviation of the weights and activations distribution as a new feature that is incorporated into the task loss function. This feature helps scale the quantizer parameter, allowing the method to take into account the full distribution of data, not just the extreme values.

**Adaptive gradient scale factor**: The impact of gradient scaling has been explored in previous methods like LSQ, which demonstrated that large gradient scales (e.g., 1) can result in weak accuracy. In our method, the standard deviation can be interpreted as an adaptive scaling factor for the quantizer parameter $\alpha$. Initially set to 1 in most of our experiments, the gradient scale is adjusted to optimize accuracy. The standard deviation as an adaptive scaling factor enables us to strike a balance between accuracy and pruning ratio by treating the gradient scale as a hyperparameter. A smaller gradient scale forces $\alpha$ to converge to smaller values, leading to less pruning, while a larger gradient scale facilitates pruning more aggressively, though potentially at the cost of accuracy. This trade-off is explored further in Section 4.5.5.

**Faster convergence**: In previous quantization methods, the clipping threshold depends solely on the quantizer parameter $\alpha$. If the data distribution changes rapidly, it may take longer for the clipping threshold to catch up with the new data distribution. In our proposed quantizer, however, the clipping threshold is influenced by both $\alpha$ and the standard deviation of the weights/activations distribution. This allows the clipping threshold to adapt more quickly to changes in the distribution, even without updating $\alpha$. As shown in Figure 4.2, this property leads to faster convergence, enabling the quantization process to better track the evolving distribution of data.

The inclusion of the standard deviation in our quantization method provides a clearer understanding of how the data is being quantized. In the original PACT quantization method, the clipping threshold (quantizer parameter $\alpha$) does not offer insight into the weights/activations distribution. In PACT, we only know that values exceeding the clipping threshold (outliers) are clipped and quantized to the highest level. In contrast, our method gives us an estimate of both outliers and pruned values by using the standard deviation of the weights/activations distribution. Though this estimation is not perfect, it provides a more nuanced understanding of the quantization process. For instance, if the weights follow a Gaussian distribution and $\alpha$ is set to 2, we can estimate that approximately 95.45% of the weights are within the clipping threshold, while the remaining 4.55% are outliers.

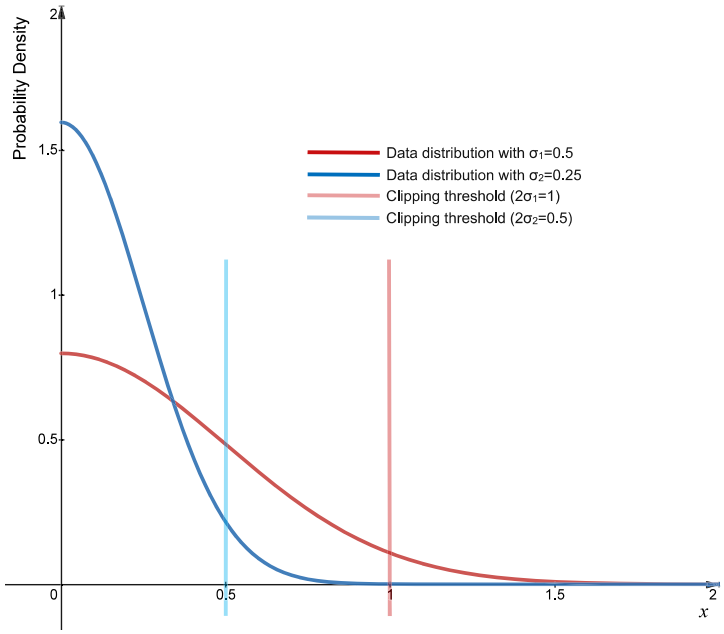



Figure 4.2: Clipping thresholds of two set of data with standard deviation of 0.5 and 0.25. The quantizer parameter $\alpha$ is fixed ($\alpha = 2$) for both data distributions.

## 4.4 Quantization Techniques

In this section, we propose two quantization techniques that can be employed to further improve the performance of our quantization method.

### 4.4.1 Improved Progressive Training

Arguably, any neural network quantized to extremely low bit-widths suffers from accuracy loss due to poor data representation. However, the inaccurate representation of the weights/activations is not the only factor negatively impacting neural networks. In fact, a major issue that causes significant performance degradation is the gradient vanishing problem, which arises from intense pruning during the quantization process [11]. Intense pruning reduces the learning capacity of neural networks, as a significant portion of the weights are removed (set to zero), which limits the network's ability to learn effectively.

Previous studies have attempted to address this issue by proposing different progressive training methods [72, 78]. For example, one recent method quantizes higher values of weights first while keeping the lower values in full precision [72]. This approach allows the gradient to backpropagate through the weights that would have been pruned due to quantization. In

another approach, it has been shown that progressive training—such as training 2-bit width networks using the trained parameters from 3-bit width networks—improves the training capacity and accuracy of quantized networks. However, simply transferring the learned parameters results in changes to the learned quantization intervals, as shown in Equation (4.4). This transformation from higher bit-width networks to lower bit-width networks often leads to further pruning due to larger pruning areas, as illustrated in Figure 4.3. By altering the quantization intervals, the learning capacity of the quantized network is reduced even more—not only because of poor data representation due to low bit-width quantization, but also because of additional pruning.

To address this issue, we propose a method to re-scale the quantizer parameter $\alpha$ so that the lower bit-width network starts with the same quantization intervals as the higher bit-width network. This is achieved by applying the following equation:

$$\alpha_b = \alpha_{b+n} \times \frac{L_b}{L_{b+n}}, \tag{4.12}$$

where $\alpha_b$ and $L_b$ represent the quantizer parameter and discretization level of the quantized networks with $b$ bits, respectively, and $b+n$ refers to the higher bit-width network. This re-scaling approach ensures that the quantization intervals of the lower bit-width network match those of the higher bit-width network at the start of training.

Our proposed progressive training method offers two key advantages: it improves the training capacity of the quantized network and prevents the quantizer from pruning parameters further. As a result, our method helps mitigate the gradient vanishing problem. More importantly, it limits the search space for the quantizer parameter $\alpha$, guiding the network to find optimal intervals close to those found in networks with higher bit-widths. This can be achieved by using smaller gradient scale values ($s$) in Equation (4.9), which enables weight decay to prevent the quantizer parameter $\alpha$ from becoming too large.

### 4.4.2 Two-Phase Training

It has been shown that a significant portion of the weights resides near the quantization interval boundaries (transition boundaries) [69]. Our experimental results confirm this observation across different networks, as shown in Figure 4.4. Based on these findings, we hypothesized that jointly optimizing the network parameters along with the quantizer's parameter could negatively impact the performance of the network. This is because a small change in the quantization intervals can lead to dramatic changes in the quantized weights,

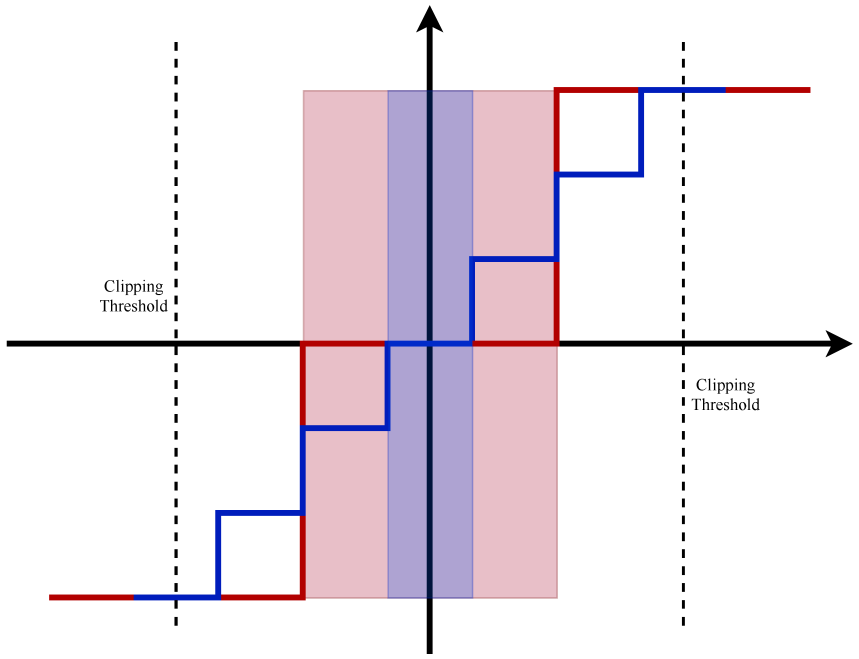



Figure 4.3: Quantization levels (intervals) of a 3- and 2-bit quantizers with the same $\alpha$. The blue line shows the intervals of the 3-bit quantizer, whereas the red line shows the intervals for the 2-bit quantizer. The pruning area of the 2-bit quantizer (rectangle with the shade of red) is 3 times wider than the 3-bit quantizer (rectangle with the shade of blue).

especially in 2-bit quantized networks where there are fewer discrete levels. Additionally, weights near the transition boundaries find it more difficult to converge to the optimal quantization intervals when the boundaries are constantly fluctuating.

Although we do not observe the same pattern in the quantized activations—due to the stabilizing effect of batch normalization—this fluctuation could still affect the activation quantization, though with less severity. Furthermore, the gradients in quantized networks with learnable parameters are influenced by the quantizer parameter when passing through the layers. From Equation (4.8), we can see that both $\alpha$ and $\sigma$ are controlling the gradients that propagate to the previous layers. Since both $\alpha$ and $\sigma$ are continuously updated during training, we suspected that this dynamic could introduce noise to the gradients.

To test this hypothesis, we retrained our quantized networks while keeping $\alpha$ and $\sigma$ frozen at their optimal values from the initial training. Interestingly, our empirical results showed consistent improvement across various networks and datasets. This outcome emphasizes the importance of our proposed two-phase training method for training parameterized quantized networks, which involves first training the network with $\alpha$ and $\sigma$ frozen at their optimal

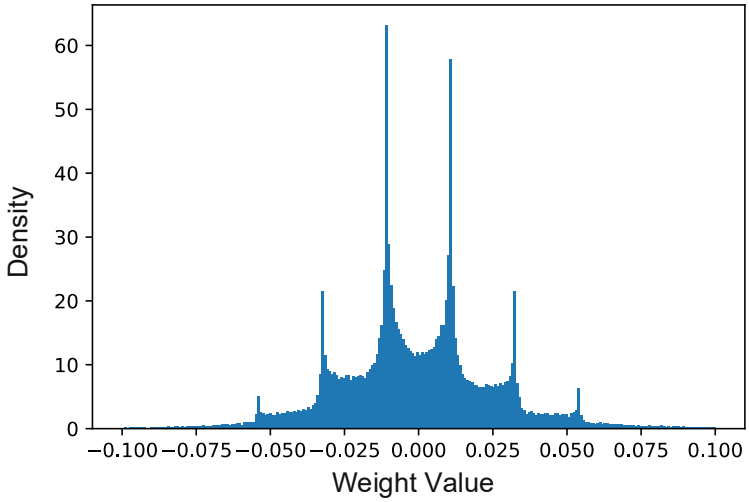



Figure 4.4: Distribution of a convolution layer from ResNet-18 model trained with our quantization method using 3 bits. The spikes in the distributions are the transition boundaries of the 3-bit quantizer.

values, and then fine-tuning them in a second phase. This approach mitigates the noise introduced by fluctuating quantization intervals and results in better overall performance.

## 4.5 Experiments

To validate our proposed quantization method, we conduct several experiments on the CIFAR-10 [81] and ImageNet [82] datasets, using various neural network architectures. We evaluate the effectiveness of our approach by comparing the performance of quantized networks under different configurations, and we perform several ablation studies to assess the impact of hyper-parameters and the proposed quantization techniques.

In all experiments, we use the same weight decay value for the quantizer's weight decay as used for the network's original weight decay. This ensures that the regularization effect of weight decay is consistent across both the network and the quantization parameter. For both the CIFAR-10 and ImageNet datasets, we adopt the same data augmentation strategy as proposed in [83], which includes techniques such as random cropping, flipping, and normalization to improve generalization and reduce overfitting.

These experiments and ablation studies help us thoroughly evaluate the performance

Table 4.1: Quantization accuracy performance on CIFAR-10 dataset with ResNet-20 model (FP accuracy: 91.74%). Methods included in this table are LQ-Net, DSQ and PACT.

| Quantization method | | Accuracy @ precision (A and W) | | | |
|---|---|---|---|---|---|
| Activations | Weights | 5 | 4 | 3 | 2 |
| DSQ [84] | DSQ | – | – | – | 90.11 |
| LQ-Net [70] | LQ-Net | – | – | 91.6 | 90.2 |
| PACT [67] | DoReFa | 91.7 | 91.3 | 91.1 | 89.7 |
| Ours | DoReFa | 92.03 | 92.00 | 91.65 | 90.32 |
| Ours | Ours | **92.27** | **92.28** | **92.23** | **90.77** |

of our quantization method, comparing it against existing techniques, and exploring how different settings affect the accuracy, pruning, and computational efficiency of the quantized networks.

### 4.5.1 ResNet-20 on CIFAR-10

To evaluate the effectiveness of our proposed quantization method, we apply it to the ResNet-20 model [83] on the CIFAR-10 dataset. We perform two sets of experiments to compare our approach with the PACT quantization method.

**Experiment 1: Progressive Quantization with Re-Scaling and Two-Phase Training**
In this experiment, we progressively quantize the ResNet-20 model by applying both re-scaling of the clipping threshold and the proposed two-phase training technique. Both weights and activations are quantized using our method. The gradient scale value $s$ is adjusted for each bit-width, and is set to {1, 1, 0.1, 0.01} for the {5, 4, 3, 2}-bit quantized ResNet-20, respectively. These gradient scale values were determined through a hyper-parameter search (see Section 4.5.6).

**Experiment 2: Comparison with PACT**
In the second experiment, we apply the weight quantization method (i.e., DoReFa [66] used in PACT and quantize ResNet-20 from scratch. Similar to PACT, we do not quantize the first and last layers of the ResNet-20 model. For activation quantization, we use a constant gradient scale of 1, independent of the bit-width.

Table 4.1 presents the quantization accuracy performance of the ResNet-20 model on the CIFAR-10 dataset, with the full precision (FP) model achieving an accuracy of 91.74%. The table compares several state-of-the-art quantization methods, including LQ-Net [70], DSQ [84],

and PACT [67], alongside the proposed method. From the results shown in Table 4.1, we observe that our quantization method outperforms PACT, even when only the activations are quantized using our approach. Moreover, the application of our two-phase training technique in conjunction with the progressive quantization method leads to further accuracy improvements, demonstrating the technique's effectiveness in reducing gradient noise. Finally, we achieve the best performance when both weights and activations are quantized using our method across all bit-widths, highlighting the superior performance of our approach.

### 4.5.2 SmallVGG on CIFAR-10

We evaluate the proposed method on the CIFAR-10 dataset by quantizing the SmallVGG network [85] under three distinct experimental setups:

**Setup-1: Quantizing Weights and Activations to 2 Bits**

In this configuration, both weights and activations of the SmallVGG network are quantized to 2 bits, with all layers being quantized except the first convolutional layer and the fully connected layer. The network's parameters are initialized using full-precision values. The model is trained for 300 epochs with a gradient scale of 0.001. The results presented in Table 4.2, demonstrate that our method not only outperforms SOTA approaches but also achieves higher accuracy compared to the full-precision model.

**Setup-2: 2-bit Activations with Binarized Weights**

In this setup, activations are quantized to 2 bits, as in Setup-1, while weights are binarized (quantized to 1 bit) using the sign function as described in [86]. Remarkably, even with binarized weights and 2-bit activations during inference, our method surpasses the performance of SOTA methods and the full-precision model, showcasing its robustness in extremely low-precision scenarios.

**Setup-3: Quantizing All Layers**

In this experiment, we extend the quantization to all layers of the SmallVGG network. However, the weights of the first convolutional layer are quantized, while its inputs remain in full precision. The model is trained under the same conditions as Setup-1. Again, our method outperforms both SOTA methods and the full-precision model.

Across all setups (both partial and full-layer quantization), the proposed method outperforms SOTA techniques (LQ-Net, HWGQ, LLSQ, RQST) in accuracy, demonstrating the robustness of the proposed quantization approach. For 2-bit activations and 2-bit weights (2/2 precision), our method achieves a maximum accuracy of 94.36%, which is higher than the accuracy of all other methods, including LQ-Net (93.50%) and LLSQ (93.31%). Quantizing all

Table 4.2: Comparison of quantization methods on CIFAR-10 using the SmallVGG network (full-precision accuracy: 93.66%). The table reports accuracy achieved at 2-bit activations (A) and 1-bit or 2-bit weights (W) under different quantization setups. "All layers" indicates whether all layers, including the first convolutional and fully connected layers, were quantized. Methods compared include LQ-Net, HWGQ, LLSQ, and RQST.

| Method | All layers | Accuracy @ Precision (A/W) | |
|---|---|---|---|
| | | 2/1 | 2/2 |
| LQ-Net [70] | No | 93.40 | 93.50 |
| HWGQ [74] | No | 92.51 | NA |
| LLSQ [76] | No | NA | 93.31 |
| Ours | No | **93.88** | **94.36** |
| RQST [85] | Yes | NA | 90.92 |
| LLSQ [76] | Yes | NA | 93.12 |
| Ours | Yes | NA | **93.90** |

layers, including the first convolution and fully connected layers, is generally more challenging, as evidenced by lower accuracy values across methods (e.g., RQST achieves 90.92%). However, our method retains high accuracy (93.90%) even when all layers are quantized, significantly outperforming RQST and slightly surpassing LLSQ (93.12%) in this challenging scenario. However, our method retains high accuracy (93.90%) even when all layers are quantized, significantly outperforming RQST and slightly surpassing LLSQ (93.12%) in this challenging scenario. When weights are binarized (1 bit) and activations are quantized to 2 bits (2/1 precision), our method achieves an accuracy of 93.88%, outperforming LQ-Net (93.40%) and HWGQ (92.51%). This indicates that the proposed approach maintains high representational power even under extreme compression conditions. The performance of our method surpasses even the full-precision baseline (93.66%) in some configurations, particularly at 2/2 precision (accuracy: 94.36%). This suggests that the quantization process introduces a regularization effect, helping the network generalize better. The results from these experiments highlight two critical observations:

- Regularization Effect of Quantization: Quantization can serve as an effective regularization mechanism, provided that the capacity of the network is not significantly reduced due to quantization.

- Over-Parameterization in SmallVGG: The results suggest that the SmallVGG network is over-parameterized for datasets like CIFAR-10, which allows for significant reduction

Table 4.3: Comparison with the existing methods using AlexNet on ImageNet (FP accuracy: 61.8%). Methods included in this table are QIL, LQ-Net, TSQ, , SYQ, PACT, LLSQ, BalancedQ, PQTSG and WEQ.

| Method | Top-1 accuracy @ precision (A and W) | | |
|---|---|---|---|
| | 4 | 3 | 2 |
| Ours | **62.5** | **62.2** | **59.2** |
| QIL [69] | 62 | 61.3 | 58.1 |
| LQ-Net [70] | – | – | 57.4 |
| TSQ [75] | – | – | 58 |
| SYQ [88] | – | – | 55.8 |
| PACT [67] | 57.2 | 55.6 | 55.0 |
| LLSQ [76] | 56.57 | 55.36 | – |
| BalancedQ [89] | – | – | 55.7 |
| PQTSG [78] | 58.1 | – | 52.5 |
| WEQ [90] | 55.9 | 54.9 | 50.6 |

in precision without degrading performance.

### 4.5.3 AlexNet on ImageNet

To evaluate the performance of our quantization method on large-scale datasets, we conducted experiments on the ImageNet dataset using a modified AlexNet architecture [87]. The modified version incorporates a batch normalization layer after each convolutional and fully-connected layer, except for the last layer.

For training 3-bit and 2-bit quantized networks, we employed the progressive training method with re-scaling of the clipping thresholds. Optimization was carried out using a cosine annealing learning rate scheduler, starting from an initial learning rate of 0.001, for a total of 70 epochs. Following standard practices in quantization, all layers were quantized except for the first convolutional layer and the final fully-connected layer. As shown in Table 4.3, our proposed quantization method outperforms previous state-of-the-art methods for low-bit quantization of AlexNet on ImageNet.

When training the 2-bit AlexNet with a gradient scale of $s = 1$, we observed a significant challenge: the training loss began to increase after several epochs, causing the network to converge to a poor local minimum. This issue arises due to extreme pruning, even when re-scaling the clipping threshold is applied. Such aggressive pruning results in a substantial reduction in the learning capacity of the network. To mitigate this problem, we reduced

Table 4.4: Top-1 accuracy of 2-bit AlexNet under different training setup.

| Setup | Gradient scale | Top-1 Acc. (%) |
|---|---|---|
| No re-scaling | 1 | 54.4 |
| Re-scaled from 4-bit | 0.01 | 58.76 |
| Re-scaled from 3-bit | 0.01 | 59.01 |
| Re-scaled from 3-bit + two-phase training | 0.01 | 59.24 |

the gradient scale to $s = 0.01$, which forced the 2-bit AlexNet to leverage the quantization intervals discovered from the 3-bit network. This adjustment reduced weight pruning and allowed the network to retain more learning capacity. Additionally, we trained the 2-bit AlexNet using the quantization intervals obtained from the 4-bit network while maintaining $s = 0.01$. Finally, we applied our proposed double-training method to further refine the 2-bit network, resulting in an additional accuracy improvement of 0.2%. As shown in Table 4.4, the 2-bit AlexNet trained with intervals from the 3-bit network achieved the best accuracy, while the model trained with intervals from the 4-bit network also surpassed the accuracy of the QIL method. These results highlight the effectiveness of progressive training with appropriate interval re-scaling and our double-training strategy in addressing the challenges of extreme low-bit quantization.

### 4.5.4 Shift-Net Results

We evaluate the effectiveness of our proposed non-uniform quantization method, described in Section 4.2.2, on the ImageNet dataset [82] using ResNet-18 and ResNet-50 models [83]. In this setup, the weights of both models are quantized into power-of-two intervals to enable efficient deployment with shift-add arithmetic. These models were specifically chosen to ensure a fair comparison of accuracy performance with state-of-the-art (SOTA) shift-add quantization methods. To optimize the networks, we employ the previously described two-phase training technique. Both ResNet-18 and ResNet-50 are initialized with pre-trained parameters from the PyTorch model zoo [91]. The networks are trained for 70 epochs using a cosine learning rate scheduler.

Table 4.5: Comparison between existing shift-add networks on ImageNet dataset. Methods included in this table are DeepShift, INQ and Sign-Sparse-Shift ($S^3$).

| Model | Method | Width | Top-1/Top-5 Acc. (%) |
|---|---|---|---|
| ResNet-18 | FP | 32 | 69.76/89.08 |
| | DeepShift [20] | 5 | 69.56/89.17 |
| | INQ [72] | 3 | 68.08/88.36 |
| | $S^3$ [11] | 3 | 69.82/89.23 |
| | Ours | 3 | **70.23/89.33** |
| | INQ | 4 | 68.89/89.01 |
| | $S^3$ | 4 | 70.47/89.93 |
| | Ours | 4 | **70.70/89.62** |
| ResNet-50 | FP | 32 | 76.13/92.86 |
| | INQ | 5 | 74.81/92.45 |
| | DeepShift | 5 | 76.33/93.05 |
| | $S^3$ | 3 | 75.75/92.80 |
| | Ours | 3 | **76.37/93.08** |

The results, as summarized in Table 4.5, demonstrate that our non-uniform quantization method achieves superior performance compared to SOTA shift-add methods for both models. On ResNet-18, our method achieves the highest accuracy for both 3-bit and 4-bit quantizations: For 3-bit weights, our method achieves 70.23% Top-1 / 89.33% Top-5 accuracy, surpassing $S^3$ (69.82% / 89.23%) and INQ (68.08% / 88.36%). For 4-bit weights, our method achieves 70.70% Top-1 / 89.62% Top-5 accuracy, outperforming $S^3$ (70.47% / 89.93%) and INQ (68.89% / 89.01%). On ResNet-50, at 3-bit precision, our method achieves 76.37% Top-1 / 93.08% Top-5 accuracy, outperforming $S^3$ (75.75% / 92.80%). Moreover, with only 3 bits, our method outperforms DeepShift, which uses 5 bits, on both ResNet-18 and ResNet-50 models.

### 4.5.5 Pruning Ratio and Accuracy Trade-off

Quantization inherently results in pruning, which can significantly reduce the memory footprint of a network. However, excessive pruning negatively impacts the model's learning capacity. In our proposed quantization method, the pruning ratio can be indirectly controlled by adjusting the gradient scale factor ($s$ in Equation (4.9)). To evaluate this property, we trained the same 3-bit quantized ResNet-18 model from the previous experiment (Section 4.5.4) on the ImageNet dataset, using various gradient scale factors. Each model was trained for 20 epochs. The results, presented in Table 4.6, reveal that slight improvements in accuracy can

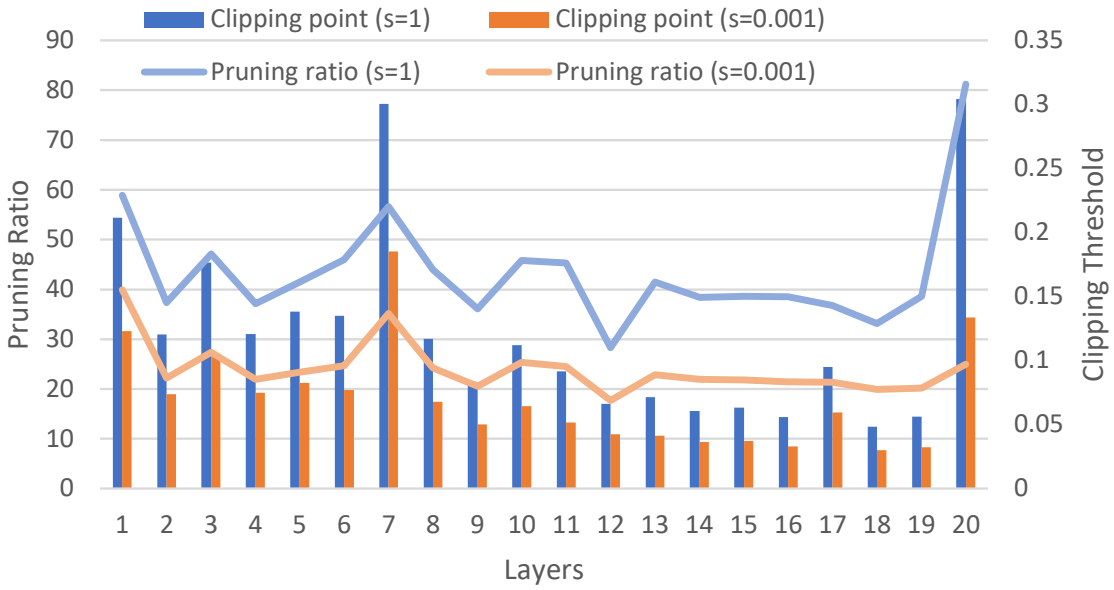



Figure 4.5: The clipping thresholds and the pruning rates of the quantized ResNet-18 weights

be achieved at the expense of a significant reduction in pruning ratio. Specifically, a 1.12% increase in Top-1 accuracy was achieved by sacrificing 18.64% of the pruning ratio. This trade-off provides flexibility to users, who can prioritize higher pruning ratios or improved accuracy depending on their application needs.

It is important to note that this trade-off is feasible only when multiple gradient scale values result in good convergence and acceptable accuracy performance. To better understand this phenomenon, we present the clipping thresholds and pruning rates of individual layers for networks trained with gradient scale values of 1 and 0.001 in Figure 4.5. As anticipated, networks trained with smaller gradient scale values exhibit smaller clipping thresholds, leading to reduced pruning rates. Conversely, larger clipping thresholds result in greater pruning ratios, as demonstrated in the results.

Table 4.6: Top-1 accuracy and pruning ratio for various gradient scale factors.

| Gradient scale | 1 | 0.1 | 0.01 | 0.001 |
|---|---|---|---|---|
| Top-1 Acc. (%) | 68.92 | 69.59 | 69.70 | 70.04 |
| Pruning ratio (%) | 40.21 | 29.74 | 24.97 | 21.57 |

### 4.5.6 Progressive Training and Re-scaling

We evaluate the effectiveness of our proposed progressive training method, specifically the re-scaling of the clipping threshold for the weights quantizer, by training the ResNet-20 model on the CIFAR-10 dataset. In this experiment, we quantize the ResNet-20 model to 2 and 3 bits. The 2-bit model is initialized and re-scaled from the 3-bit and 4-bit models, while the 3-bit model is initialized and re-scaled using the 4-bit model.

As shown in Table 4.7, the 2-bit model achieves higher accuracy when the clipping threshold is re-scaled from the 3-bit model. Interestingly, both the 2-bit and 3-bit models produce comparable accuracy, even when the quantizer parameter ($\alpha$) is not updated (i.e., the gradient scale is set to 0). This implies that it is possible to use the same quantization intervals derived from higher-bit networks in lower-bit networks and still obtain satisfactory performance. This observation highlights the importance and effectiveness of our proposed re-scaling technique. Additionally, examining the results from columns where the gradient scale is set to 0.1 and 0.01, we find that updating $\alpha$ with a smaller gradient scale does not always lead to better performance. This suggests that there are diminishing returns in terms of accuracy improvements when $\alpha$ is adjusted too aggressively with small gradient scales.

Table 4.7: Accuracy performance of ResNet-20 on CIFAR-10, quantized using 2 and 3 bits with different clipping threshold initialization and gradient scale values.

| Bit-width | Clipping Threshold Initialization | Gradient scale | | | |
|---|---|---|---|---|---|
| | | 1 | 0.1 | 0.01 | 0 |
| 2 | From 3 | 88.06 | 90.34 | 90.69 | 90.12 |
| 2 | From 4 | 87.55 | 90.29 | 90.11 | 89.57 |
| 3 | From 4 | 92.14 | 92.23 | 92.07 | 92.10 |

## 4.6 Discussion

Here we discuss the differences between our shift-based neural network and the shift-based neural networks listed in Table 4.5: DeepShift [20], INQ [72] and Sign-Sparse-Shift ($S^3$) [11].

INQ method [72] achieves quantization through a stepwise process that includes three interdependent operations: weight partitioning, group-wise quantization, and re-training. Weight partitioning divides the weights in each layer of a pre-trained full-precision model into two disjoint groups, each with complementary roles in the quantization process. The first

group is quantized using a rounding method, while the second group compensates for the accuracy loss introduced by quantization through re-training. These operations are iteratively applied to the second group of weights until all weights are quantized to powers of two or zero. While the INQ method achieves promising results, it is limited to pre-trained models and cannot be applied during training. To enable power-of-two quantization during training, the DeepShift method [20] reparametrizes weights into two elements: a 2-bit sign-operator parameter and a bit-shift parameter. The sign-operator, a ternary value, determines the shift process (no shift, shift-add, or shift-subtract), while the bit-shift parameter specifies the number of bit-wise shifts. DeepShift can be applied to both pre-trained models and models trained from scratch, with both parameters updated through backpropagation. The Sign-Sparse-Shift ($S^3$) method [11] extends DeepShift by learning each individual bit in the sign-operator and bit-shift parameters, thereby improving accuracy. However, the $S^3$ method significantly increases memory requirements, as each bit of the quantized weights becomes a learnable parameter, posing scalability challenges for large neural networks.

In contrast to these methods, our shift-based neural network directly quantizes weights without reparametrization. The proposed method introduces only a minimal number of additional parameters (e.g., parameter $\alpha$) during training, resulting in a negligible impact on the overall network size and memory usage.

Compared to the $S^3$ method, our approach offers several distinct advantages. First, it enables initialization with full-precision pre-trained model parameters, accelerating convergence significantly. While the $S^3$ method requires 200 epochs to achieve competitive results, our method achieves superior accuracy within just 20 epochs, offering a 10x faster convergence rate. Furthermore, in an $N$-bit shift-based neural network, the $S^3$ method necessitates $N$ times the number of parameters during training, leading to increased memory usage and higher hardware resource requirements. In contrast, our method introduces only minor overhead, limited to the parameters $\alpha$ and the standard deviation of the weights, thereby minimizing the memory footprint while maintaining substantial performance gains.

## 4.7 Conclusion

In this chapter, we introduced a novel quantization method that leverages the distribution of weights and activations during the quantization process. By utilizing the standard deviation of these parameters, our method outperforms existing quantization techniques across several image classification tasks. To further enhance the performance, we proposed two training strategies. The first, a two-phase training technique, mitigates gradient noise and the

fluctuations of the quantizer's transition boundaries that result from jointly optimizing the network and quantizer parameters. The second strategy, a re-scaling technique, addresses the gradient vanishing problem and improves the training of quantized networks. Our approach provides flexibility, allowing users to balance accuracy and network size by adjusting the pruning ratio.

To achieve the primary objective of this study, we introduce a training framework for non-uniform quantization, specifically base-2 logarithmic quantization, in which weights are quantized into power-of-two intervals. This approach replaces complex multipliers with efficient shift-add operations, significantly reducing computational cost. Our method improves the training of quantized networks by decreasing both training time (number of epochs) and memory utilization (number of parameters). Notably, our approach achieves competitive performance after 20 epochs using only 3 bits on the ImageNet dataset, demonstrating its effectiveness in real-world applications.

# 5

# Towards Lossless ANN-to-SNN Conversion

## 5.1 Introduction

Spiking neural networks (SNNs) are a biologically inspired class of artificial neural networks that process information through discrete spiking events, rather than continuous activations as in traditional artificial neural networks (ANNs) [92]. These networks emulate the dynamics of biological neurons, where a neuron emits a spike when its membrane potential exceeds a threshold. This behavior is often modeled using frameworks such as the leaky integrate-and-fire (LIF) model [93] or the Hodgkin-Huxley equations [94].

SNNs are increasingly recognized for their potential in energy-efficient computation, particularly when deployed on neuromorphic hardware such as Intel's Loihi [95], IBM's TrueNorth [96], and SpiNNaker [97]. This energy efficiency arises from the sparse nature of spike-based communication and the event-driven computational paradigm, making SNNs well-suited for resource-constrained and real-time applications [21]. The development of neuromorphic hardware has catalyzed the adoption of SNNs in energy-constrained environments. SNNs have demonstrated efficacy in a range of tasks, including event-based vision [98],

robotics [23], and sensor data processing [99], particularly in scenarios requiring real-time computation. SNNs are uniquely suited for tasks involving temporal or spatio-temporal data due to their intrinsic ability to encode and process information in both spatial and temporal domains. Applications include speech recognition [100], dynamic gesture detection [22], and motion tracking [101].

Initial SNN training methodologies predominantly relied on unsupervised learning approaches, including Hebbian learning [102] and spike-timing-dependent plasticity (STDP) [103]. However, recent developments in supervised learning have facilitated the application of gradient-based optimization techniques. Surrogate gradient methods, which approximate the non-differentiable spike function with a continuous surrogate, have been particularly impactful, enabling backpropagation through time (BPTT) and enhancing the scalability of SNN training [24–26]. The discrete and non-differentiable nature of spike events complicates direct optimization using standard backpropagation. While surrogate gradient methods offer a practical solution, they introduce approximation errors and require careful parameter tuning [104]. Moreover, surrogate gradient methods encounter significant computational demands and reduced efficiency during the training process, especially when applied to complex network architectures [30, 105]. Addressing this trade-off requires alternative training methodologies.

A significant body of work focuses on converting pre-trained ANNs into SNNs. This approach utilizes the firing rate of spiking neurons to approximate the continuous activations of ANNs. ANN-to-SNN conversion methods have emerged as a popular approach for leveraging the task-specific performance of ANNs while utilizing the sparse, event-driven computational nature of SNNs. These methods typically adapt activation-based ANNs to spike-based frameworks by normalizing weights [28], rescaling thresholds [30, 106], and mitigating conversion losses [31, 32]. While significant progress has been made, key challenges remain, particularly in addressing the trade-off between inference latency and accuracy.

The initial research on ANN-to-SNN conversion was introduced by Cao et al. [27], and later expanded by Diehl et al. [28], who improved the conversion results through data-based and model-based normalization. To handle more complex datasets and deeper network architectures, Rueckauer et al. [29, 107] and Sengupta et al. [106] proposed advanced scaling methods, including weight normalization and threshold rescaling, by analyzing the relationship between the activations of ANNs and the spike rates of SNNs. However, a key limitation of these early methods is the requirement for hundreds to thousands of simulation time steps to achieve high accuracy. This dependency arises from conversion biases, which are particularly

pronounced in deeper networks [30–32, 108].

To address the limitations of traditional ANN-to-SNN conversions, several methods have been proposed to mitigate conversion loss and reduce simulation length. Rueckauer et al. [107] introduced the reset-by-subtraction mechanism, which preserves residual information by reducing the membrane potential by the threshold voltage after a spike, instead of resetting it to a fixed resting potential. This approach addresses potential information loss during resets [109]. Rueckauer et al. [29] proposed the use of percentile-based thresholds to avoid outliers in activation distributions, while Sengupta et al. [106] recommended scaling thresholds to normalize activations more effectively. Adjusting thresholds based on input-output spike frequencies, further improved the accuracy of converted SNNs [109]. Techniques such as channel-level threshold balancing [110], time steps-aware threshold optimization [30], and bias-shifting strategies [31, 32] have been applied to address discrepancies in activation frequencies across neurons in the same layer, improving information transmission in shorter simulations. Despite these advances, ANN-to-SNN conversion methods face persistent challenges:

- **Trade-Off Between Accuracy and Latency:** While improvements in scaling and threshold adjustment techniques have reduced simulation lengths, achieving near-original ANN accuracy with ultra-low latency (e.g., fewer than 4 time steps) remains difficult. This trade-off is a fundamental constraint due to the discrete nature of spiking computations.

- **Discretization Limitations:** The uniform discretization of numerical inputs across neurons in the same layer fails to account for variations in activation frequencies. As a result, some neurons struggle to transmit information effectively in short simulation sequences, necessitating longer simulation lengths for high accuracy (Deng et al., 2020).

In this study, we propose a novel framework for lossless ANN-to-SNN conversion, focusing on overcoming the discretization limitations inherent in existing methods through the adoption of an improved quantization technique. Specifically, we conduct a comprehensive analysis of the conversion bias in the proposed framework and demonstrate that the it is zero for any arbitrary number of discrete quantization intervals in the source ANN and any arbitrary number of time steps in the converted SNN, provided that the number of time steps is at least equal to or greater than the number of quantization intervals. The effectiveness of our approach is validated by achieving ANN-equivalent accuracy with a reduction of up to $2\times$ in time steps compared to state-of-the-art methods.

## 5.2 Preliminaries

In this section, we introduce the basic framework for ANN-to-SNN conversion.

### 5.2.1 Neuron Model for ANN

In the $\ell^{th}$ layer of an ANN, the post-activation neurons, denoted as $\mathbf{a}_\ell^{post}$, are described by the following relationship:

$$\mathbf{a}_\ell^{post} = h(\mathbf{a}_\ell^{pre}), \quad \mathbf{a}_\ell^{pre} = \mathbf{W}_\ell \ \mathbf{a}_{\ell-1}^{post}, \tag{5.1}$$

where $\mathbf{a}_\ell^{pre}$ represents the pre-activation state of the neurons in the $\ell^{th}$ layer, $\mathbf{W}_\ell$ is the weight matrix, and $h(\cdot)$ is the ReLU activation function.

### 5.2.2 Neuron Model for SNN

For the SNN, we employ the Integrate-and-Fire (IF) neuron model [27]. The pre-spike (similar to the pre-activation in ANN) membrane potential $\mathbf{m}_\ell^{pre}(t)$ of neurons in the $\ell^{th}$ layer at time step $t$ is updated as:

$$\mathbf{m}_\ell^{pre}(t) = \tau \mathbf{m}_\ell^{post}(t-1) + \mathbf{W}_\ell \ \mathbf{s}_{\ell-1}(t), \tag{5.2}$$

where $\tau$ is the leaky factor, $\mathbf{s}_{\ell-1}(t)$ is the spike output vector from neurons in layer $\ell-1$ at time $t$, and $\mathbf{m}_\ell^{post}(t-1)$ is the post-spike (similar to the post-activation in ANN) potential at the preceding time step $(t-1)$. In the entirety of this study, we assume $\tau$ is always 1. A neuron generates a spike when its membrane potential exceeds the firing threshold $v_{th}$, after which the membrane potential is reset according to a hard-reset mechanism:

$$\mathbf{s}_\ell(t) = \begin{cases} 1 & \text{if } \ \mathbf{m}_\ell^{pre}(t) >= v_{th} \\ 0 & \text{otherwise} \end{cases}, \tag{5.3}$$

$$\mathbf{m}_\ell^{post}(t) = \mathbf{m}_\ell^{pre}(t) \cdot (1 - \mathbf{s}_\ell(t)). \tag{5.4}$$

This hard-reset approach can lead to information loss [29, 109]. To mitigate this, we adopt a soft-reset mechanism, where the membrane potential is reduced by the threshold $v_{th}$ upon

firing [107]. Under this model, the equations become:

$$\mathbf{s}_\ell(t) = \begin{cases} v_{th} & \text{if } \mathbf{m}_\ell^{pre}(t) >= \mathbf{v}_{th} \\ 0 & \text{otherwise} \end{cases}, \tag{5.5}$$

$$\mathbf{m}_\ell^{post}(t) = \mathbf{m}_\ell^{pre}(t) - \mathbf{s}_\ell(t)\mathbf{v}_{th}. \tag{5.6}$$

To elucidate the distinctions between hard-reset and soft-reset mechanisms, consider two neurons, $m1$ employing a hard-reset mechanism and $m2$ employing a soft-reset mechanism. Both neurons receive three consecutive weighted input spikes of 1.25 at time steps $T1$, $T2$ and $T3$, with a firing threshold $v_{th} = 0.5$. Since the weighted spikes exceed $v_{th}$ neuron $m1$ fires a spike at each time step $T1$, $T2$ and $T3$, resetting its potential to zero after each spike. In contrast, neuron $m2$ fires spikes for seven consecutive time steps ($T1$ to $T7$); with each spike, its potential decreases by 0.5. The pre- and post-spike potentials of neurons $m1$ and $m2$ are depicted in Figure 5.1.

### 5.2.3 ANN-to-SNN conversion

The objective of ANN-to-SNN conversion is to align the spiking neuron firing rates in the SNN with the continuous activation values in the ANN after $T$ time steps:

$$\mathbf{a}_\ell^{post} \simeq \frac{1}{T}\sum_{t=0}^{T} \mathbf{s}_\ell(t) \tag{5.7}$$

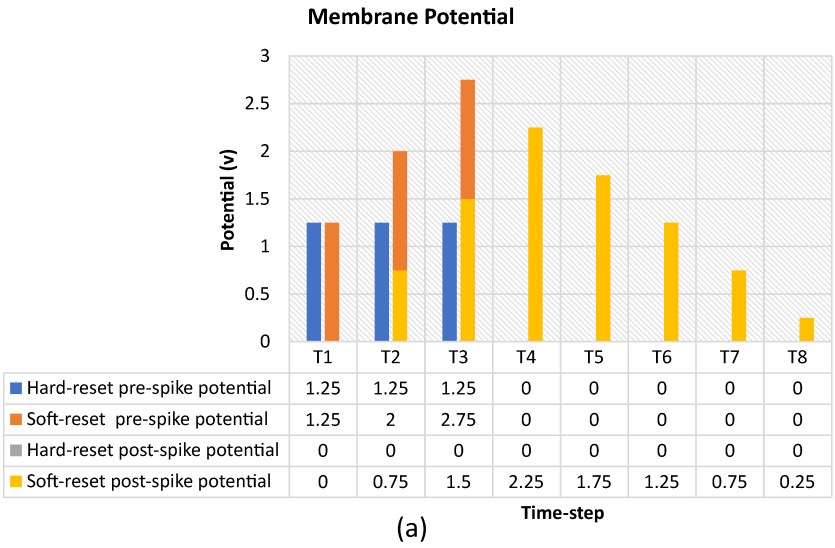
Let $\overline{\mathbf{m}}_\ell$ denote the average pre-spike potential over $T$ time steps:

$$\overline{\mathbf{m}}_\ell = \frac{1}{T}\sum_{t=0}^{T} \mathbf{m}_\ell^{pre}(t) \tag{5.8}$$

Based on the firing condition Equation (5.5), and the average pre-spike potential Equation (5.8), the firing rate (i.e., average post-spike output) can be expressed as:

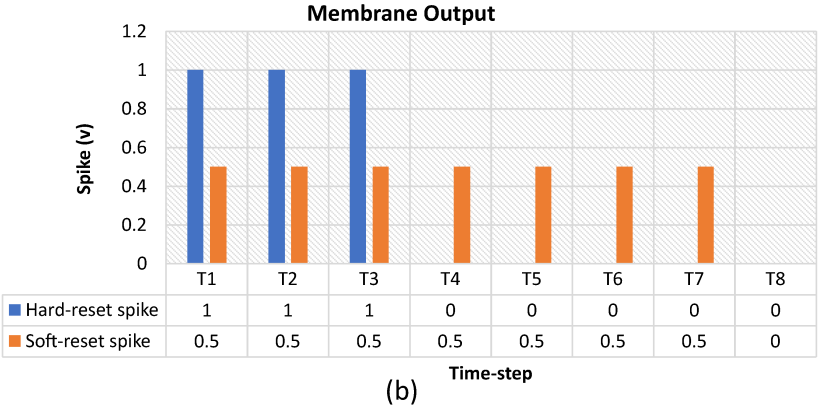
$$\frac{1}{T}\sum_{t=0}^{T} \mathbf{s}_\ell(t) = \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_\ell.T}{v_{th}} \right\rfloor, \tag{5.9}$$

where $\lfloor\cdot\rfloor$ denotes the floor function. According to Equation (5.9), when the number of time steps $T \to \infty$ and firing threshold $v_{th} = \max(\mathbf{a}_\ell^{post})$, the firing rates of spiking neurons




| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|---|---|---|---|---|---|---|---|---|
| Hard-reset pre-spike potential | 1.25 | 1.25 | 1.25 | 0 | 0 | 0 | 0 | 0 |
| Soft-reset pre-spike potential | 1.25 | 2 | 2.75 | 0 | 0 | 0 | 0 | 0 |
| Hard-reset post-spike potential | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Soft-reset post-spike potential | 0 | 0.75 | 1.5 | 2.25 | 1.75 | 1.25 | 0.75 | 0.25 |

(a)




| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|---|---|---|---|---|---|---|---|---|
| Hard-reset spike | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Soft-reset spike | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0 |

(b)

Figure 5.1: (a) Pre- and post-spike potentials. (b) Generated spikes

.

converge to the continuous activation values. This principle forms the foundation of early conversion methods [27–29, 106, 107], which required thousands of time steps to achieve accuracy comparable to ANNs. In recent studies, the conversion of quantized ANNs has been proposed as a method to reduce the number of time steps required. By replacing the activation function in the ANN with a floor quantization mechanism, the post-activation output is redefined as:

$$\mathbf{a}_\ell^{post} = \frac{\lambda}{L}\text{clip}(\left\lfloor \frac{\mathbf{a}_\ell^{pre}.L}{\lambda} \right\rfloor, 0, L), \tag{5.10}$$

where $L$ is the number of quantization intervals (steps), and $\lambda$ is the activation clipping threshold. According to Equation (5.7), Equation (5.9) and Equation (5.10), by equating the

quantization steps $L$ and clipping threshold $\lambda$ in the ANN with the time steps $T$ and firing threshold $v_{th}$ in the SNN, the equivalence between the two models is achieved:

$$\frac{\lambda}{L}\text{clip}(\left\lfloor \frac{\mathbf{a}_\ell^{pre}.L}{\lambda} \right\rfloor, 0, L) = \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_\ell.T}{v_{th}} \right\rfloor. \tag{5.11}$$

## 5.3 Conversion Error

As established in Equation (5.11), the conversion error, defined as $\mathbf{a}_\ell^{post} - \frac{1}{T}\sum_{t=0}^{T}\mathbf{s}_\ell(t)$, becomes zero under the assumptions $L = T$, $\lambda = v_{th}$. However, in practical applications of SNNs, the number of time steps $T$ is often variable. Consequently, achieving a zero conversion error necessitates training distinct ANNs, each tailored to a specific set of quantization steps. Even if this limitation is acknowledged, quantization bias introduced by the flooring function persist, adversely impacting the performance of converted SNNs, particularly when operating with an extremely low number of time steps $T$. In the following subsections, we provide an analysis of the conversion error when $L \neq T$ as well as the quantization bias.

### 5.3.1 Analysis of Quantization Bias introduced by the Flooring Function

We begin by analyzing the quantization bias introduced by the flooring function. To estimate this bias, we calculate the average quantization error by computing the expected value of the difference between the original input $x$ and its quantized counterpart. The quantization bias is mathematically expressed as:

$$\begin{aligned} Bias_Q &= \mathbb{E}(x - \frac{\lambda}{L}\text{clip}(\left\lfloor \frac{x \cdot L}{\lambda} \right\rfloor, 0, L)) \\ &= \mathbb{E}(x) - \frac{\lambda}{L}\mathbb{E}(\text{clip}(\left\lfloor \frac{x \cdot L}{\lambda} \right\rfloor, 0, L)), \end{aligned} \tag{5.12}$$

where $\lambda$ is the clipping threshold, $L$ is the number of quantization steps, and clip($\cdot$,0,$L$) limits the range of the quantized value. To isolate and focus solely on the bias introduced by the flooring function, we assume that $x$ is also bounded by $\lambda$. This assumption simplifies the analysis by ensuring that the input values lie within the range defined by the clipping threshold, thereby allowing us to concentrate on the impact of quantization. Thus, we redefine the quantization bias to explicitly account for the assumption that $x$ is bounded by $\lambda$. The revised quantization bias is expressed as:

$$Bias_Q = \mathbb{E}(x) - \frac{\lambda}{L}\mathbb{E}(\left\lfloor \frac{x \cdot L}{\lambda} \right\rfloor), \tag{5.13}$$

where $x \in [0, \lambda]$, ensuring the analysis focuses exclusively on the flooring function's contribution to the bias. Furthermore, we assume that $x$ is uniformly distributed within the interval $[0, \lambda]$. This assumption allows for a straightforward computation of the quantization bias (average quantization error) by leveraging the uniform distribution properties. Since $x \sim U[0, \lambda]$, the expected value of x denoted as $\mathbb{E}(x)$ can be computed directly. For a uniform distribution over $x \in [0, \lambda]$, the expected value is the midpoint of the range:

$$\mathbb{E}(x) = \frac{0 + \lambda}{2} = \frac{\lambda}{2}. \tag{5.14}$$

To compute the expected value of $\lfloor \frac{xL}{\lambda} \rfloor$, where $x \sim U[0, \lambda]$, we proceed as follows:

Let $y = \frac{xL}{\lambda}$. Since $x \sim U(0, \lambda)$, the variable y is uniformly distributed over $(0, L)$. The task reduces to determining the expected value of $\lfloor y \rfloor$, where $y \sim U(0, L)$. The flooring function maps $y$ to the greatest integer $k \in \{0, 1, ..., L-1\}$ such that $k <= y$. For each $k$, the probability that $\lfloor y \rfloor = k$ is determined by the length of the interval over which $y$ satisfies this condition:

$$\lfloor y \rfloor = k \quad \text{if} \quad y \in [k, k+1)$$

Given that $y$ is uniformly distributed over$[0, L]$, the probability $P(\lfloor y \rfloor = k)$ is proportional to the length of the interval $[k, k+1)$, normalized by the total range $L$:

$$P(\lfloor y \rfloor = k) = \frac{\text{Length of the interval where } \lfloor y \rfloor = k}{L}. \tag{5.15}$$

For $k \in \{1, ..., L-1\}$, the interval length is 1.0, resulting in:

$$P(\lfloor y \rfloor = k) = \frac{1}{L}, \quad k \in \{1, ..., L-1\}. \tag{5.16}$$

The expected value of $\lfloor y \rfloor$ is then given by:

$$\mathbb{E}(\lfloor y \rfloor) = \sum_{k=0}^{L-1} k.P(\lfloor y \rfloor = k). \tag{5.17}$$

By substituting the probabilities corresponding to each value of $k$:

$$\begin{aligned}\mathbb{E}(\lfloor y \rfloor) &= \sum_{k=1}^{L-1} k.\frac{1}{L} \\ &= \frac{1}{L}\sum_{k=1}^{L-1} k,\end{aligned} \tag{5.18}$$

where summation $\sum_{k=1}^{L-1} k$ is a standard arithmetic series and evaluates to $\frac{(L-1)L}{2}$. Thus:

$$\begin{aligned}\mathbb{E}(\lfloor y \rfloor) = \mathbb{E}(\left\lfloor \frac{x \cdot L}{\lambda} \right\rfloor) &= \frac{1}{L} \cdot \frac{(L-1)L}{2} \\ &= \frac{L-1}{2}\end{aligned} \tag{5.19}$$

By combining the results from Equation (5.13), Equation (5.14) and Equation (5.19), the quantization bias is derived as follows:

$$\begin{aligned}Bias_Q &= \frac{\lambda}{2} - \frac{\lambda}{L} \cdot \frac{L-1}{2} \\ &= \frac{\lambda}{2}(1 - \frac{L-1}{L}) \\ &= \frac{\lambda}{2L}\end{aligned} \tag{5.20}$$

According to Equation (5.20), the quantization process using the flooring function systematically underestimates the input, thereby introducing a downward bias. This occurs because the quantized value is always less than or equal to the original value. To mitigate the quantization bias introduced by the flooring function, it is necessary to employ either very large quantization steps $L$, an extremely small clipping threshold $\lambda$, or a combination of both. However, this approach renders the flooring function a suboptimal choice for the ANN-to-SNN conversion framework.

### 5.3.2 Analysis of Conversion Bias Due to Mismatched $L$ and $T$

To analyze the the conversion error that arises when $L \neq T$, we calculate the conversion bias (the average conversion error) introduced by both the quantization with the flooring function and the firing condition of the spiking neurons. Using Equation (5.11), we define

the conversion bias as:

$$Bias_C = \mathbb{E}(\frac{\lambda}{L}\text{clip}(\left\lfloor \frac{\mathbf{a}_\ell^{pre} \cdot L}{\lambda} \right\rfloor, 0, L)) - \mathbb{E}(\frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_\ell \cdot T}{v_{th}} \right\rfloor). \tag{5.21}$$

By incorporating Equation (5.19) into Equation (5.21), the conversion error can be expressed as:

$$Bias_C = \frac{\lambda}{L} \cdot \frac{L-1}{2} - \frac{v_{th}}{T} \cdot \frac{T-1}{2}. \tag{5.22}$$

Given $\lambda = v_{th}$, Equation (5.22) simplifies to:

$$Bias_C = \frac{\lambda}{2}(\frac{L-1}{L} - \frac{T-1}{T}). \tag{5.23}$$

The formulation in Equation (5.23), demonstrates that any mismatch between the quantization steps $L$ and the time steps $T$ introduces a conversion bias, particularly when $T >> L$. This bias can accumulate and propagate through the network, adversely affecting the performance of the converted SNN, as supported by our experimental results (see Figure 5.2). However, Equation (5.23) also indicates that for sufficiently large values of $L$ and $T$, the conversion bias approaches zero. This observation suggests that it is feasible to use different values for $L$ and $T$ without significantly impacting performance, a claim further validated by our experimental results (see Figure 5.4).

## 5.4 Lossless ANN-to-SNN Conversion

In the preceding section, we examined the conversion and quantization biases arising from the use of the flooring function in the source ANN. In this section, we address the limitations of the flooring function by proposing the use of a rounding function as an alternative. As with the flooring function, we analyze the quantization bias associated with the rounding function and demonstrate that it introduces no biases provided that the input is uniformly distributed. To enable the conversion of the source ANN utilizing the rounding function, we introduce a bias term for the converted SNN. We further analyze the conversion bias in our proposed ANN-to-SNN conversion method and establish that it is independent of the quantization steps $L$ and time steps $T$. Notably, we show that the conversion bias is zero under this approach. Lastly, we integrate our standard deviation-based quantization method, introduced in Chapter 4, to determine the optimal clipping threshold $\lambda$, while minimizing the number of required quantization steps $L$.

### 5.4.1 Analysis of Quantization Bias Introduced by the Rounding Function

For analyzing the quantization bias we follow the same steps introduced in Section 5.3.1. The quantization bias is expressed as:

$$Bias_Q = \mathbb{E}(x) - \frac{\lambda}{L}\mathbb{E}(\left\lfloor \frac{x \cdot L}{\lambda} \right\rceil), \tag{5.24}$$

where $\lfloor \cdot \rceil$ is the rounding function.

To compute the expected value of $\left\lfloor \frac{xL}{\lambda} \right\rceil$, where $x \sim U[0, \theta]$, we proceed as follows:
Let $y = \frac{xL}{\lambda}$. Since $x \sim U[0, \lambda]$, the variable $y$ is uniformly distributed over $[0, L]$. The task reduces to determining the expected value $\lfloor y \rceil$, where $y \sim U[0, L]$. The rounding function maps $y$ to the nearest integer $k \in \{0, 1, ..., L\}$. For each $k$, the probability that $\lfloor y \rceil = k$ is determined by the length of the interval over which $y$ satisfies these conditions:

- $y$ rounds to $k$ if $y \in [k - 0.5, k + 0.5)$, except at the boundaries,
- $y \in [0, 0.5)$ for $k = 0$,
- $y \in [L - 0.5, L]$ for $k = L$,

Given that $y$ is uniformly distributed over $[0, L]$, the probabilities $P(\lfloor y \rceil = k)$ are proportional to the interval lengths, normalized by the total range $L$:

$$P(\text{round}(y) = k) = \frac{\text{Length of the interval where } \lfloor y \rceil = k}{L}. \tag{5.25}$$

Therefore, the probabilities of $\lfloor y \rceil = k$ for different $k$ values are:

- for $k = 0$, interval length is 0.5, thus $P(\lfloor y \rceil = 0) = \frac{0.5}{L}$,
- for $k = L$, interval length is 0.5, thus $P(\lfloor y \rceil = L) = \frac{0.5}{L}$,
- for $k \in \{1, ..., L - 1\}$, interval length is 1.0, thus $P(\lfloor y \rceil = k) = \frac{1}{L}$

The expected value of $\lfloor y \rceil$ is then given by:

$$\mathbb{E}(\lfloor y \rceil) = \sum_{k=0}^{L} k \cdot P(\lfloor y \rceil = k). \tag{5.26}$$

By substituting the probabilities corresponding to each value of $k$:

$$\begin{aligned}\mathbb{E}(\lfloor y \rceil) &= 0 \cdot \frac{0.5}{L} + (\sum_{k=1}^{L-1} k \cdot \frac{1}{L}) + L \cdot \frac{0.5}{L} \\ &= 0.5 + \frac{1}{L} \sum_{k=1}^{L-1} k,\end{aligned} \tag{5.27}$$

where summation $\sum_{k=1}^{L-1} k$ evaluates to $\frac{(L-1)L}{2}$. Thus:

$$\begin{aligned}\mathbb{E}(\lfloor y \rceil) &= 0.5 + \frac{1}{L} \cdot \frac{(L-1)L}{2} \\ &= 0.5 + \frac{L-1}{2} \\ &= \frac{L}{2}.\end{aligned} \tag{5.28}$$

By combining the results from Equation (5.24), Equation (5.19) and Equation (5.14), the quantization bias is derived as follows:

$$Bias_Q = \frac{\lambda}{2} - \frac{\lambda}{L} \cdot \frac{L}{2} = 0. \tag{5.29}$$

As indicated by Equation (5.29), the rounding function, in contrast to the flooring function, eliminates the need for selecting excessively large quantization steps ($L$) or extremely small clipping thresholds ($\lambda$) to minimize quantization bias, as the bias is inherently zero. It is important to note that, in practical scenarios where input data is not uniformly distributed, quantization bias can still persist even when using the rounding function. However, unlike the flooring function, the rounding function does not introduce a systematically downward bias, thereby offering a distinct advantage in terms of quantization performance.

### 5.4.2 ANN-to-SNN Conversion with Round Quantization Mechanism

To enable ANN-to-SNN conversion using the round quantization mechanism, it is necessary to revise the average post-spike output of converted SNN in Equation (5.9) and the the post-activation output of the source ANN Equation (5.10) by substituting the flooring function with the rounding function. The post-activation output in Equation (5.10) with the rounding

function is redefined as:

$$\mathbf{a}_{\ell}^{post} = \frac{\lambda}{L}\text{clip}(\left\lfloor \frac{\mathbf{a}_{\ell}^{pre}.L}{\lambda} \right\rfloor, 0, L), \tag{5.30}$$

However, directly replacing the flooring function with the rounding function for the average post-spike output in Equation (5.9) is not feasible due to the constraints imposed by the firing condition in Equation (5.5), which necessitates the use of the flooring function. Nevertheless, since the rounding function can be expressed as $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$, it is possible to replace the flooring function with the rounding function in Equation (5.9) by applying a leftward shift of $\frac{1}{2}$. This shift can be incorporated into the spiking neuron as a bias term, referred to as the initial membrane potential at time zero ($\mathbf{m}_{\ell}^{pre}(t=0)$):

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T}\mathbf{s}_{\ell}(t) &= \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_{\ell}.T}{v_{th}} + \frac{1}{2} \right\rfloor \\
&= \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_{\ell}\cdot T + \frac{1}{2}v_{th}}{v_{th}} \right\rfloor \\
&= \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_{\ell}\cdot T + \mathbf{m}_{\ell}^{pre}(t=0)}{v_{th}} \right\rfloor \\
&= \frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_{\ell}\cdot T}{v_{th}} \right\rfloor
\end{aligned} \tag{5.31}$$

### 5.4.3 Analysis of Conversion Bias with the Rounding Function

Here, we provide a formal analysis of the conversion bias of the proposed round quantization mechanism. The conversion bias is mathematically formulated as:

$$Bias_C = \mathbb{E}(\frac{\lambda}{L}\text{clip}(\left\lfloor \frac{\mathbf{a}_{\ell}^{pre}.L}{\lambda} \right\rfloor, 0, L)) - \mathbb{E}(\frac{v_{th}}{T}\left\lfloor \frac{\overline{\mathbf{m}}_{\ell}.T}{v_{th}} \right\rfloor). \tag{5.32}$$

By incorporating Equation (5.28) into Equation (5.32), the conversion bias can be expressed as:

$$\begin{aligned}
Bias_C &= \frac{\lambda}{L}\cdot\frac{L}{2} - \frac{v_{th}}{T}\cdot\frac{T}{2} \\
&= \frac{\lambda}{2} - \frac{v_{th}}{2}.
\end{aligned} \tag{5.33}$$

Given $\lambda = v_{th}$, Equation (5.33) simplifies to:

$$Bias_C = \frac{\lambda}{2} - \frac{\lambda}{2} = 0. \tag{5.34}$$

As demonstrated in Equation (5.34), the conversion bias in the proposed ANN-to-SNN conversion method is entirely independent of the quantization steps $L$ and the time steps $T$.

### 5.4.4 Optimal Clipping and Firing Threshold

For SNNs to be practical, they must achieve accuracy performance equivalent to ANNs within a limited number of time steps. Achieving this requires the source ANN to be quantized using extremely small quantization steps ($L$), while preserving its accuracy.

A significant challenge arises because most modern deep ANNs employ batch normalization prior to the activation function, causing a substantial portion of the pre-activation values to cluster near zero. Consequently, quantizing these pre-activations with a small number of quantization steps results in a large proportion of values being mapped to the first quantization interval, which corresponds to zero. To address this problem, it is essential to carefully select the clipping threshold $\lambda$ to minimize the information loss introduced by the quantization process. However, determining an optimal clipping threshold is not straightforward [30]. In Chapter 4, we introduced the standard deviation (std)-based quantization method, which effectively quantizes ANNs by learning the optimal clipping threshold for a specified number of quantization steps. Since the firing threshold ($v_{th}$) of the converted SNN is directly derived from the clipping threshold, our std-based quantization method can be leveraged to train source ANNs with minimal quantization steps while simultaneously determining the optimal clipping threshold for each layer. Specifically, the clipping threshold in Equation (5.30) is replaced with the product of the standard deviation of the pre-activations ($\sigma(\mathbf{a}_\ell^{pre})$) and a trainable parameter ($\alpha$), as expressed in the following equation:

$$\mathbf{a}_\ell^{post} = \frac{\sigma(\mathbf{a}_\ell^{pre}) \cdot \alpha_\ell}{L} \text{clip}(\left\lfloor \frac{\mathbf{a}_\ell^{pre}.L}{\sigma(\mathbf{a}_\ell^{pre}) \cdot \alpha_\ell} \right\rfloor, 0, L). \tag{5.35}$$

The optimal value of the parameter $\alpha$ is determined during the training of the source ANN using gradient descent and backpropagation. Detailed explanations of the backpropagation process are provided in Chapter 4.

### 5.4.5 Conversion Framework

Here, we detail our proposed ANN-to-SNN conversion process.

**Architecture Modification:** An SNN can only be converted from the ANNs, whose behavior can be accurately replicated by the converted SNN. For example, SNNs are unable to simulate the max-pooling function, as it necessitates prior knowledge of which neuron will fire with the highest intensity in advance. Therefore, such functions must be replaced with alternatives whose behavior is time-independent in SNNs. In our implementation, we address this limitation by replacing max-pooling layers with average-pooling layers.

**Source ANN Quantization:** In our proposed conversion framework, the source ANNs are quantized using the rounding function. To identify the optimal clipping threshold $\lambda$, which serves as the firing threshold $v_{th}$ in the converted SNNs, we utilize our standard deviation-based quantization method, as detailed in Chapter 4. The number of quantization steps is determined to ensure that the accuracy of the quantized ANNs matches that of the original continuous ANNs. To streamline the quantization process, the pre-trained models are directly quantized.

**Weight Rescaling, Batch Normalization and Initial Bias:** In Equation (5.5), when the firing condition is satisfied, the spiking neuron generates an output equal to $v_{th}$. However, in practical applications of SNNs, the spiking neurons must produce only binary outputs (i.e., "0" or "1"). To achieve this, the trained weight parameters from the quantized ANNs are rescaled by the value of $v_{th}$ before being transferred to the converted SNNs. This rescaling enables the spiking neurons to generate binary spikes. Furthermore, to eliminate the final remaining multiplication operation in SNNs, which occurs in batch normalization (BN), we deconstruct the BN function and integrate its parameters into the weights and biases of the corresponding layer. Finally, the initial membrane potential is adjusted by adding $\mathbf{m}^{pre}\ell(t=0) = \frac{1}{2}v_{th}$ to the biases:

$$W_{SNN} \leftarrow W_{ANN} \cdot v_{th} \cdot \frac{\gamma_{BN}}{\sigma_{BN}}, \qquad b_{SNN} \leftarrow \frac{1}{2}v_{th} + \beta_{BN} + (b_{ANN} - \mu_{BN})\frac{\gamma_{BN}}{\sigma_{BN}}. \tag{5.36}$$

Algorithm 7 provides a summary of the proposed ANN-to-SNN conversion framework.

## 5.5 Experiments

The effectiveness of the proposed ANN-SNN conversion methodology is evaluated on the CIFAR-10 [81] and ImageNet [82] datasets, utilizing the ResNet-20, and ResNet-34 [83] architectures.

For the CIFAR-10 dataset, data augmentation techniques, including normalization, horizontal flipping, and random cropping, are employed. ResNet-20 model is trained on CIFAR-10

for 90 epochs. The initial learning rate is set to 0.1 and is reduced by a factor of 0.1 every 30 epochs. Various quantization levels, $L \in \{1, 2, 3, 4, 8, 12, 16\}$, are utilized to train the source ANN. A similar data augmentation strategy is applied to the ImageNet dataset. Training on ImageNet is conducted exclusively with the ResNet-34 architecture and employs quantization level $L = 8$. The source ANN model is initialized using pre-trained weights and subsequently fine-tuned for 20 epochs at each quantization level. The initial learning rate is set to 0.001, and a cosine decay scheduler is implemented to adjust the learning rate during training.

### 5.5.1 Flooring Versus Rounding Functions

In Section 5.4, we establish that the conversion bias introduced by the rounding function is zero provided that $\lambda = v_{th}$ and $\mathbf{m}_\ell^{pre}(t = 0)$, irrespective of whether $L$ and $T$ are equivalent. To validate this finding, we train a ResNet-20 model on the CIFAR-10 dataset as the source ANN, using both flooring and rounding functions with $L = 4$. The corresponding converted SNNs are then evaluated over various time steps $T$. As illustrated in Figure 5.2, when the source ANN is quantized using the rounding function, the accuracy of the converted SNN aligns with that of the source ANN at $T = 8$ and surpasses it for $T > 16$. It is important to note that, while the rounding function introduces no conversion bias, the presence of unexpected spikes still contributes to a residual bias [31]. These unanticipated spikes arise due to the time-sensitive behavior of spiking neurons. For example, in a situation where the average pre-spike potential of a neuron is lower than the firing threshold, the corresponding output in the source ANN would be zero, leading to the expectation that the neuron would not fire in the SNN. However, there are instances where the neuron's pre-spike potential may exceed the threshold at specific time steps, triggering a spike despite the average pre-spike potential remaining sub-threshold. This phenomenon is evident at $L = 4$, where the converted

**Algorithm 7:** ANN-to-SNN Conversion

**input :** Pre-trained ANN model, Quantization step $L$
**output :** Converted SNN model

Replace Max-Pooling layers with Avgerage-pooling layers
Learn $\{W_{ANN}, b_{ANN}, \lambda\ \beta_{BN}, \gamma_{BN}, \sigma_{BN}, \mu_{BN}\}$ by quantizing ANN with $L$ quantization steps using std-based quantization method
$v_{th} \leftarrow \lambda$
$W_{SNN} \leftarrow W_{ANN} \cdot v_{th} \cdot \frac{\gamma_{BN}}{\sigma_{BN}}$
$b_{SNN} \leftarrow \frac{1}{2} v_{th} + \beta_{BN} + (b_{ANN} - \mu_{BN}) \frac{\gamma_{BN}}{\sigma_{BN}}$
**return** SNN

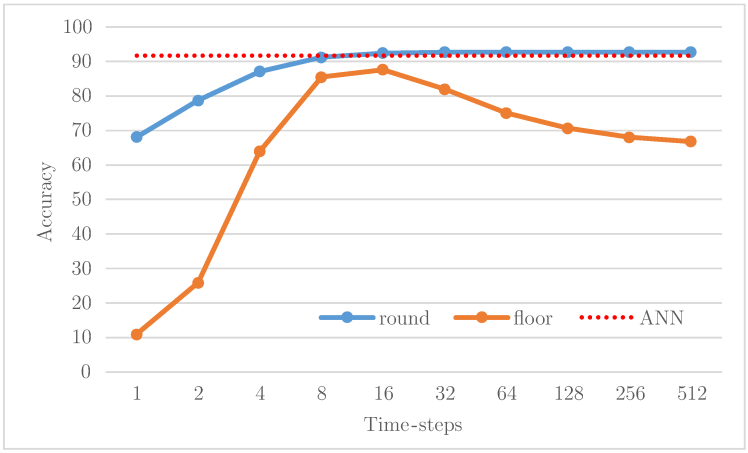



Figure 5.2: Accuracy performance of the converted SNNs from the source ANNs quantized with flooring and rounding functions

SNN fails to achieve the expected accuracy of the source ANN. However, as the number of time steps increases ($T > 4$), the impact of the conversion bias due to unexpected spikes becomes negligible. In contrast, the accuracy of the converted SNN derived from the source ANN quantized with the flooring function peaks at $T = 16$ but declines as $T$ increases beyond this point ($T > 16$). This decline in accuracy can be attributed to the conversion bias induced by the flooring function when $L \neq T$. As demonstrated in Equation (5.23), the fractional term $\frac{T-1}{T}$ grows larger with increasing $T$. Consequently, when $L$ remains constant, as in this experiment where $L = 4$, the conversion bias increases with larger $T$.

### 5.5.2 Ablation Study on the Impact of Quantization Steps

We investigate the impact of the number of quantization steps $L$, on the accuracy performance of the converted SNNs across different time steps $T$. In the source ANN quantized using $L$ intervals, the parameter $L$ directly influences the accuracy of the quantized ANN. Specifically, as the number of quantization steps increases, the accuracy of the quantized ANN improves correspondingly. Moreover, achieving comparable accuracy in the SNNs converted from these quantized ANNs necessitates setting the number of time steps, $T$, to values at least equal to $L$ (i.e., $T \geq L$). Based on these considerations, the following conditions can be established for selecting an appropriate number of quantization steps:

- $L$ must be sufficiently large to ensure the desired accuracy performance of the quantized ANN.
- $L$ must be sufficiently small to enable the converted SNN to achieve acceptable accuracy

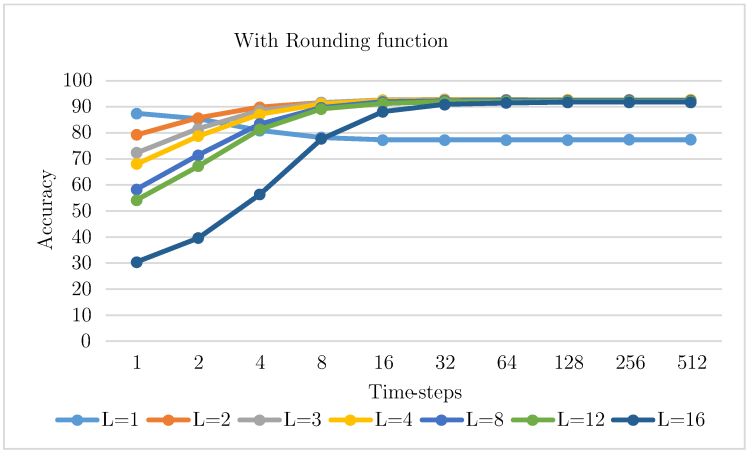



Figure 5.3: Accuracy performance of the converted SNNs from ANNs quantized with rounding function across different quantization steps $L$ and time steps $T$

with the minimum possible number of time steps, $T$, where $T \geq L$.

These criteria highlight the trade-offs between quantization resolution (accuracy) and temporal requirements (latency) for accurate SNN performance. To better analyze the impact of $L$ on accuracy performance and the number of time steps $T$, we train the ResNet-20 model on CIFAR-10 datasets using both rounding and flooring functions with different quantization steps $L \in \{1, 2, 3, 4, 8, 12, 16\}$. The accuracy performance of the converted SNNs are reported in Table 5.1 and Table 5.2, and illustrated in Figure 5.3 and Figure 5.4.

From Figure 5.3 and Table 5.1, we observe that for time steps $T \leq 8$, the SNNs converted from ANNs with fewer quantization steps ($L \leq 4$) exhibit higher accuracy compared to those converted from ANNs with a larger number of quantization steps ($L > 4$). For instance, when $L = 2$, the accuracy of the quantized ANN is 90.71%, which is 1.01% lower than that of the full-precision ANN. However, the accuracy of the converted SNN at $T = 8$ improves to

Table 5.1: Impact of different quantization steps $L$ on the accuracy performance of the SNNs converted from the ANNs quantized with rounding function across different time steps $T$.

| Quantization steps | ANN | $T = 1$ | $T = 2$ | $T = 4$ | $T = 8$ | $T = 16$ | $T = 32$ | $T = 64$ | $T = 128$ | $T = 256$ | $T = 512$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| L=1 | 87.46 | 87.45 | 85.39 | 80.87 | 78.24 | 77.3 | 77.24 | 77.22 | 77.23 | 77.33 | 77.33 |
| L=2 | 90.71 | 79.23 | 85.66 | 89.86 | 91.68 | 92.18 | 92.43 | 92.42 | 92.5 | 92.50 | 92.51 |
| L=3 | 91.84 | 72.34 | 81.55 | 88.51 | 91.68 | 92.57 | 92.87 | 92.74 | 92.7 | 92.70 | 92.68 |
| L=4 | 92.37 | 68.08 | 78.74 | 87.11 | 91.16 | 92.45 | 92.69 | 92.68 | 92.69 | 92.71 | 92.72 |
| L=8 | 92.39 | 58.33 | 71.36 | 83.38 | 89.66 | 91.86 | 92.32 | 92.49 | 92.45 | 92.49 | 92.45 |
| L=12 | 92.13 | 54.14 | 67.19 | 81.32 | 89.14 | 91.33 | 91.94 | 91.99 | 92.04 | 92.06 | 92.11 |
| L=16 | 91.74 | 30.36 | 39.63 | 56.35 | 77.67 | 88.11 | 90.87 | 91.45 | 91.73 | 91.74 | 91.76 |

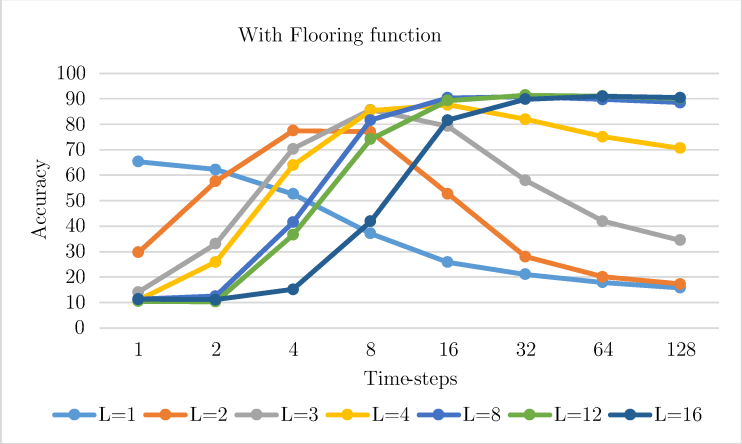



Figure 5.4: Accuracy performance of the converted SNNs from ANNs quantized with flooring function across different quantization steps $L$ and time steps $T$

91.68%, reducing the gap to just 0.06% compared to the full-precision ANN. Interestingly, as $L$ increases, a slight degradation in accuracy performance is noted. This phenomenon can be attributed to the role of quantization as a form of regularization for neural networks. Specifically, smaller quantization steps introduce a regularizing effect by approximating the original network less precisely. Conversely, as $L$ increases, the quantized network becomes a closer approximation of the original network, thereby diminishing the regularization effect and potentially leading to reduced generalization performance.

The results for SNNs converted from ANNs quantized with flooring function (as shown in Table 5.2 and Figure 5.4) indicate that accuracy performance declines significantly when both the quantization step ($L$) and the time step ($T$) are small. Additionally, in SNNs with smaller quantization steps ($L <= 4$), accuracy performance exhibits a substantial decrease when the ratio $\frac{T}{L}$ exceeds 4. In contrast, when both $L$ and $T$ are sufficiently large

Table 5.2: Impact of different quantization steps $L$ on the accuracy performance of the SNNs converted from the ANNs quantized with flooring function across different time steps $T$.

| Quantization steps | ANN | $T = 1$ | $T = 2$ | $T = 4$ | $T = 8$ | $T = 16$ | $T = 32$ | $T = 64$ | $T = 128$ |
|---|---|---|---|---|---|---|---|---|---|
| L=1 | 65.33 | 65.33 | 62.26 | 52.61 | 37.08 | 25.82 | 21.09 | 17.88 | 15.77 |
| L=2 | 87.97 | 29.69 | 57.56 | 77.47 | 77.09 | 52.67 | 28.03 | 20.08 | 17.31 |
| L=3 | 91.43 | 14.11 | 33.08 | 70.35 | 85.7 | 79.28 | 58.04 | 41.96 | 34.48 |
| L=4 | 92.16 | 10.91 | 25.90 | 63.94 | 85.44 | 87.62 | 81.97 | 75.08 | 70.61 |
| L=8 | 92.31 | 11.34 | 12.50 | 41.48 | 81.62 | 90.39 | 90.92 | 89.76 | 88.47 |
| L=12 | 92.15 | 10.47 | 10.32 | 36.54 | 74.16 | 89.33 | 91.44 | 91.07 | 90.16 |
| L=16 | 91.90 | 11.27 | 11.06 | 15.19 | 41.89 | 81.68 | 89.86 | 90.95 | 90.44 |

Table 5.3: Comparison with existing method on ResNet-20 and CIFAR-10 dataset

| Method | ANN | $T = 2$ | $T = 4$ | $T = 8$ | $T = 16$ | $T = 32$ | $T = 64$ | $T > 64$ |
|---|---|---|---|---|---|---|---|---|
| SN [106] | 89.10 | | | | | | | 87.56 |
| HT [105] | 93.15 | | | | | | | 92.22 |
| RMP [109] | 91.47 | | | | | | | 91.36 |
| TCL [108] | 92.26 | | | | | | | 92.06 |
| TSC [111] | 91.47 | | | | | | 69.38 | 91.42 |
| RTS [32] | 93.61 | | | | 92.41 | 93.30 | 93.55 | 93.56 |
| SNNC-AP [30] | 95.46 | | | | | 94.78 | 95.30 | 95.45 |
| QCFS [31] | 91.77 | 73.20 | 83.75 | 89.55 | 91.62 | 92.24 | 92.35 | 92.41 |
| Ours | 91.74 | 85.66 | 89.86 | 91.68 | 92.57 | 92.87 | 92.58 | 92.72 |

(e.g., $L > 4$ and $T > 16$), no significant decline in accuracy performance is observed. As discussed in Section 5.3.2, the conversion bias associated with the flooring function is given by $\frac{\lambda}{2}\left|\frac{T-1}{T} - \frac{L-1}{L}\right|$. This relationship clearly demonstrates that the conversion bias diminishes as both $T$ and $L$ increase, resulting in improved accuracy performance for SNNs converted from ANNs using the flooring function.

A comparison of the quantized ANN accuracies presented in Table 5.1 and Table 5.2 reveals a clear advantage of quantization using the rounding function over the flooring function, particularly at extremely low quantization steps ($L < 4$). As discussed in Section 5.4.4 quantizing pre-activations with a small number of quantization steps results in a significant proportion of values being mapped to the first quantization interval, which corresponds to zero. This effect is exacerbated when using the flooring function, as the length of the first quantization interval is twice that of the rounding function.

### 5.5.3 Comparison with the Existing Work

We compare our proposed method with existing approaches on the CIFAR-10 dataset using ResNet-20 and the ImageNet dataset using ResNet-34. The accuracy results for ResNet-20, presented in Table 5.3, demonstrate that our method outperforms all existing approaches across different time steps. At time step $T = 2$, our method achieves an accuracy that is 12.46% higher than the QCFS method. At $T = 8$, our accuracy is only 0.06% lower than that of the source ANN, whereas QCFS requires 16 time steps to achieve comparable accuracy. Furthermore, it is noteworthy that other methods require a minimum of 64 time steps to align the performance of their SNNs with that of their source ANNs.

As shown by the accuracy results in Table 5.4, our proposed conversion method surpasses

Table 5.4: Comparison with existing method on ResNet-34 and ImageNet dataset

| Method | ANN | $T = 8$ | $T = 16$ | $T = 32$ | $T = 64$ | $T = 128$ | $T = 256$ | $T > 256$ |
|---|---|---|---|---|---|---|---|---|
| SN [106] | 70.69 | | | | | | | 65.47 |
| TCL [108] | 70.87 | | | | | | | 70.66 |
| HT [105] | 70.20 | | | | | | 61.48 | 65.10 |
| RMP [109] | 70.64 | | | | | | 55.65 | 69.89 |
| TSC [111] | 70.64 | | | | | | 55.65 | 69.93 |
| RTS [32] | 75.66 | | | 0.09 | 0.12 | 3.19 | 47.11 | 75.08 |
| SNNC-AP [30] | 75.66 | | | 64.54 | 71.12 | 73.45 | 74.61 | 75.45 |
| QCFS [31] | 74.32 | | 59.35 | 69.37 | 72.35 | 73.15 | 73.37 | 73.39 |
| Ours | 73.36 | 54.33 | 68.93 | 71.81 | 72.30 | 72.34 | 72.52 | 72.55 |

existing methods on the ImageNet dataset while utilizing significantly fewer time steps. Specifically, our framework achieves a 2× reduction in time steps compared to existing approaches without compromising accuracy. For example, the QCFS and SNNC-AP methods require 128 and 256 time steps, respectively, to achieve a 1% accuracy gap between the source ANN and the converted SNN, whereas our method achieves this with only $T = 64$ time steps. Furthermore, at ultra-low time steps $T = 16$, our framework demonstrates remarkable performance, with accuracy falling only 4.43% below that of the source ANN. These findings underscore the effectiveness of the proposed framework in overcoming the limitations of existing methods, such as high latency and computational overhead, while maintaining competitive performance on large-scale datasets like ImageNet.

## 5.6 Discussion and Conclusion

In this chapter, we present a novel ANN-to-SNN conversion framework capable of achieving accuracy equivalent to ANNs while utilizing 50% fewer time steps compared to state-of-the-art methods on CIFAR-10 and ImageNet datasets. The proposed framework employs a rounding function during the quantization of source ANNs, which we demonstrate ensures a zero conversion bias, irrespective of the number of quantization steps or time steps. Additionally, we conduct a comprehensive analysis of the conversion bias when the flooring function is used for quantization. Furthermore, we employ the std-based quantization method to determine the optimal firing threshold, enabling the quantization of source ANNs with the minimum number of quantization steps. This approach facilitates the use of fewer time steps in the converted SNNs. Although we mathematically demonstrate that the conversion bias in the

proposed method is zero, a mismatch between the accuracy of the source ANNs and the converted SNNs is still observed. It is hypothesized that an additional source of conversion bias arises from the unexpected firing of neurons in SNNs [31]. These unintended spikes occur due to the time-dependent nature of spiking neurons. For instance, consider a scenario where the average pre-spike potential of a neuron is below the firing threshold. In the source ANN, this would result in a zero output, leading us to expect that the corresponding neuron will not fire in the SNN. However, it is possible for the pre-spike potential of the neuron to exceed the firing threshold at certain time steps, causing it to fire despite its average pre-spike potential remaining below the threshold. Addressing and mitigating this source of conversion bias, resulting from the unexpected firing of spiking neurons, requires further investigation and is an avenue for future work.

# 6

# Conclusions and Future Work

Multiplierless neural networks have been introduced as an alternative to conventional neural networks to address the high power consumption and computational overhead caused by multiplication operations. These networks replace multipliers with hardware-efficient operations, such as bit-wise logical operations (e.g., AND, OR), additions, bit-shifts, and comparisons. The elimination of multipliers significantly reduces the memory footprint and energy consumption of these models, making them particularly well-suited for real-time applications on resource-constrained devices, such as mobile platforms, embedded systems, and neuromorphic hardware. However, despite these advantages, multiplierless neural networks often suffer from accuracy degradation. For instance, stochastic computing (SC)-based neural networks and spiking neural networks (SNNs) require hundreds to thousands of time steps to achieve comparable accuracy to their binary-radix counterparts. This increased latency negates the energy efficiency gains from removing multipliers. Similarly, shift-based neural networks require careful quantization and optimization to preserve the accuracy performance of full-precision models. This dissertation addresses these limitations by eliminating sources of accuracy degradation in SC-based, spiking, and shift-based neural networks.

In Chapter 3, we propose the dynamic sign-magnitude (DSM) stochastic stream to improve the representation of near-zero values, which are often inaccurately represented by bipolar

stochastic streams. The DSM approach enhances the precision of short-sequence SC-based multiplication using XNOR operations. By adopting DSM, the sequence length required for SC-based neural networks is reduced by a factor of 64 while maintaining accuracy levels comparable to state-of-the-art techniques.

In Chapter 4, we present training framework for base-2 logarithmic quantization of neural networks, where weights are quantized to powers of two (i.e., $\pm 2^n$). This framework employs the standard deviation of weights to determine the optimal clipping threshold, thereby eliminating quantization outliers using backpropagation. This method ensures a robust quantization scheme while minimizing accuracy loss.

In Chapter 5, we mathematically analyze the performance degradation in converted SNNs and identify the conversion error as a result of quantization using the flooring function during ANN-to-SNN conversion. To address this issue, we propose an improved ANN-to-SNN conversion framework that employs the rounding function during weight quantization. We demonstrate that this approach achieves zero conversion error, irrespective of the quantization levels or time steps. Furthermore, we integrate the logarithmic quantization method from Chapter 4 to reduce the number of quantization steps in source ANNs. This optimization allows the converted SNNs to operate with ultra-low time steps. The proposed ANN-to-SNN framework achieves accuracy equivalent to the source ANNs while reducing the number of time steps by 50% compared to state-of-the-art methods on CIFAR-10 and ImageNet datasets. These results pave the way for ultra-low latency and energy-efficient neural network implementations on edge devices.

In addition to the aforementioned improvements, Chapter 2 introduces a novel multiplierless neural network design referred to as the FSM-based network. We demonstrate that FSM-based networks can synthesize complex multi-input functions, such as 2D Gabor filters, and perform non-sequential tasks, such as image classification, on stochastic bit streams without requiring multiplications. The FSM-based network operates using look-up tables (LUTs) alone. Furthermore, the proposed FSM-based model is capable of addressing temporal tasks. Unlike long short-term memory (LSTM) networks, the FSM-based model's required storage for training is independent of the number of time steps. This unique property enables FSM-based networks to learn extremely long data dependencies while achieving significant resource savings, including:

- A reduction in storage required for intermediate training values by a factor of $l\times$,
- A 33% decrease in power consumption during training, and

- A reduction in inference operations by a factor of 7×.

Through these contributions, this dissertation addresses the challenges of accuracy degradation, latency, and computational efficiency in multiplierless neural networks. The proposed methods not only enhance the performance of SC-based, shift-based, and spiking neural networks but also introduce innovative FSM-based designs that enable efficient learning and inference for temporal and non-temporal tasks. These advancements bring multiplierless neural networks closer to practical deployment in real-world, resource-constrained environments.

## 6.1 Suggestions for Future work

In this dissertation, we proposed several methods and implementations to enhance the accuracy and reduce the latency of multiplierless neural networks. However, further research is needed to refine these designs and enable their deployment in real-world applications, particularly on edge devices. Below, we outline several potential directions for future work on multiplierless neural network designs:

**Benchmarking and Real-World Applications**

- **Objective**: Evaluate multiplierless neural networks on real-world tasks to demonstrate their practical utility.
- **Examples**:
  - Deploying multiplierless models for edge AI applications, such as IoT devices, robotics, and mobile vision systems.
  - Benchmarking the proposed FSM-based and SC-based neural networks performance on large-scale datasets, such as ImageNet to validate scalability.
  - Benchmarking the proposed ANN-to-SNN conversion method performance neuromorphic datasets, such as DVS-Gesture [98] and DVS-CIFAR10 [112] to validate scalability.

**Integration with Advanced Training Techniques**

- **Objective**: Adapt modern training strategies to improve the performance of multiplierless neural networks.
- **Examples**:

  - Leveraging knowledge distillation to transfer accuracy from full-precision models to multiplierless versions.
  - Applying pruning and sparsification to further reduce computation and energy consumption.

**Theoretical Analysis of Multiplierless Networks**

- **Objective**: Establish theoretical foundations to analyze and improve the convergence, robustness, and expressiveness of multiplierless networks.
- **Examples**:
  - Investigating the limits of accuracy and capacity of bit-shift and logical operation-based networks.
  - Studying the impact of multiplierless operations on gradient propagation and optimization during training.

**Hybrid Models with Multiplierless Layers**

- **Objective**: Explore architectures that combine multiplierless and full-precision layers for improved efficiency and accuracy.
- **Examples**:
  - Using full-precision multiplications in critical layers (e.g., input or final layers) while employing multiplierless operations in hidden layers.
  - Adaptive layer-wise precision switching during inference to balance energy and accuracy trade-offs.

**Energy-Efficient Hardware Implementations**

- **Objective**: Develop dedicated hardware accelerators optimized for multiplierless computations.
- **Examples**:
  - FPGA- or ASIC-based architectures tailored for bit-wise logical operations and shift-based arithmetic.

  - Designing systolic arrays or neuromorphic hardware that efficiently map multiplierless operations in SNNs.
  - Hardware-aware neural network design techniques to optimize performance on low-power platforms.

**Eliminating Conversion Error in SNNs**

- **Objective**: Develop a theoretical framework to analyze and mitigate conversion errors arising from unexpected neuron firing in SNNs.
- **Examples**:
  - Analyzing the effect of firing thresholds on conversion errors caused by unexpected neuron spikes.
  - Designing optimization and calibration techniques for weights and biases to suppress unintended spikes.
  - Exploring the influence of various input encoding methods on the occurrence of unexpected firing events.

# Bibliography


[1] A. Ardakani, A. Ardakani, and W. J. Gross, "Training binarized neural networks using ternary multipliers," *IEEE Design & Test*, vol. 38, no. 6, pp. 44–52, 2021.

[2] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[3] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[4] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[5] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-based systems*, vol. 212, p. 106622, 2021.

[6] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[7] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1638–1647.

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, cite arxiv:1704.04861.

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[10] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A survey of stochastic computing neural networks for machine learning applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 7, pp. 2809–2824, 2020.

[11] X. Li, B. Liu, Y. Yu, W. Liu, C. Xu, and V. Partovi Nia, "S ̃3: Sign-sparse-shift reparametrization for effective training of low-bit shift networks," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[12] J. D. Nunes, M. Carvalho, D. Carneiro, and J. S. Cardoso, "Spiking neural networks: A survey," *IEEE Access*, vol. 10, pp. 60 738–60 764, 2022.

[13] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *CoRR*, vol. abs/1512.08571, 2015.

[14] S. Stanton, P. Izmailov, P. Kirichenko, A. A. Alemi, and A. G. Wilson, "Does knowledge distillation really work?" *Advances in Neural Information Processing Systems*, vol. 34, pp. 6906–6919, 2021.

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

[16] W. J. Gross and V. C. Gaudet, *Stochastic Computing: Techniques and Applications*. Springer, 2019.

[17] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A stochastic computational multi-layer perceptron with backward propagation," *IEEE Transactions on Computers*, vol. 67, no. 9, pp. 1273–1286, 2018.

[18] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE transactions on computers*, vol. 60, no. 1, pp. 93–105, 2010.

[19] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI implementation of deep neural networks using integral stochastic computing," in *2016 9th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, Sept 2016, pp. 216–220.

[20] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "Deepshift: Towards multiplication-less neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 2359–2368.

[21] K. Roy, A. Jaiswal, and P. Panda, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, no. 7784, pp. 607–617, 2019.

[22] S. Singh, A. Sarma, S. Lu, A. Sengupta, V. Narayanan, and C. R. Das, "Gesture-snn: Co-optimizing accuracy, latency and energy of snns for neuromorphic vision sensors," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2021, pp. 1–6.

[23] 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.

[24] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, p. 331, 2018.

[25] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," *Advances in neural information processing systems*, vol. 31, 2018.

[26] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 1311–1318.

[27] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, pp. 54–66, 2015.

[28] 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)*. ieee, 2015, pp. 1–8.

[29] 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, p. 682, 2017.

[30] Y. Li, S. Deng, X. Dong, R. Gong, and S. Gu, "A free lunch from ann: Towards efficient, accurate spiking neural networks calibration," in *International conference on machine learning*. PMLR, 2021, pp. 6316–6325.

[31] T. Bu, W. Fang, J. Ding, P. DAI, Z. Yu, and T. Huang, "Optimal ANN-SNN conversion for high-accuracy and ultra-low-latency spiking neural networks," in *International Conference on Learning Representations*, 2022.

[32] S. Deng and S. Gu, "Optimal conversion of conventional artificial neural networks to spiking neural networks," in *International Conference on Learning Representations*, 2021.

[33] A. Ardakani, A. Ardakani, and W. Gross, "Training linear finite-state machines," *Advances in Neural Information Processing Systems*, vol. 33, pp. 7173–7183, 2020.

[34] A. Ardakani, A. Ardakani, B. Meyer, J. J. Clark, and W. J. Gross, "Standard deviation-based quantization for deep neural networks," *arXiv preprint arXiv:2202.12422*, 2022.

[35] A. Ardakani, A. Ardakani, and W. J. Gross, "A regression-based method to synthesize complex arithmetic computations on stochastic streams," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[36] A. Ardakani, Z. Ji, A. Ardakani, and W. Gross, "The Synthesis of XNOR Recurrent Neural Networks with Stochastic Logic," in *Thirty-third Conference on Neural Information Processing Systems*, 2019.

[37] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2012, pp. 480–487.

[38] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan, "Logical computation on stochastic bit streams with linear finite-state machines," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1474–1486, 2014.

[39] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[40] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design with CD-ROM*, 2nd ed. USA: McGraw-Hill, Inc., 2004.

[41] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[42] B. Gaines, "Stochastic Computing Systems," in *Advances in Information Systems Science*, ser. Advances in Information Systems Science, J. Tou, Ed. Springer US, 1969, pp. 37–172.

[43] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 92:1–92:19, May 2013.

[44] B. D. Brown and H. C. Card, "Stochastic neural computation i: Computational elements," *IEEE Trans. Comput.*, vol. 50, no. 9, pp. 891–905, Sep. 2001.

[45] N. Onizawa, D. Katagiri, K. Matsumiya, W. J. Gross, and T. Hanyu, "Gabor filter based on stochastic computation," *IEEE Signal Processing Letters*, vol. 22, no. 9, pp. 1224–1228, Sep. 2015.

[46] G. Arfken, m. Hans-Jürgen Weber, H. Weber, and F. Harris, *Mathematical Methods for Physicists*. Elsevier, 2005.

[47] J. Mutch and D. G. Lowe, "Object class recognition and localization using sparse features with limited receptive fields," *Int. J. Comput. Vision*, vol. 80, no. 1, p. 45–57, Oct. 2008.

[48] S. Liu, H. Jiang, L. Liu, and J. Han, "Gradient descent using stochastic circuits for efficient training of learning machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2530–2541, 2018.

[49] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2688–2699, Oct 2017.

[50] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a Large Annotated Corpus of English: The Penn Treebank," *Comput. Linguist.*, vol. 19, no. 2, pp. 313–330, Jun. 1993.

[51] A. Karpathy, J. Johnson, and F.-F. Li, "Visualizing and Understanding Recurrent Networks," *CoRR*, vol. abs/1506.02078, 2015.

[52] N. Onizawa, D. Katagiri, K. Matsumiya, W. J. Gross, and T. Hanyu, "An accuracy/energy-flexible configurable gabor-filter chip based on stochastic computation with dynamic voltage–frequency–length scaling," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 3, pp. 444–453, 2018.

[53] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 405–418, 2017.

[54] J. Li, A. Ren, Z. Li, C. Ding, B. Yuan, Q. Qiu, and Y. Wang, "Towards acceleration of deep convolutional neural networks using stochastic computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 115–120.

[55] A. Zhakatayev, S. Lee, H. Sim, and J. Lee, "Sign-magnitude sc: getting 10x accuracy for free in stochastic computing for deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[56] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[57] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[58] A. Ren, Z. Li, Y. Wang, Q. Qiu, and B. Yuan, "Designing reconfigurable large-scale deep learning systems using stochastic computing," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 2016, pp. 1–7.

[59] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in neural information processing systems*, 2017, pp. 1509–1519.

[60] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin, "Shiftaddnet: A hardware-inspired deep network," *arXiv preprint arXiv:2010.12785*, 2020.

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

[62] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[63] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "Vlsi implementation of deep neural network using integral stochastic computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2688–2699, 2017.

[64] S. Ghosh-Dastidar and H. Adeli, "Spiking neural networks," *International journal of neural systems*, vol. 19, no. 04, pp. 295–308, 2009.

[65] S. C. Smithson, K. Boga, A. Ardakani, B. H. Meyer, and W. J. Gross, "Stochastic computing can improve upon digital spiking neural networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 309–314.

[66] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[67] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[68] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," *arXiv preprint arXiv:1902.08153*, 2019.

[69] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4350–4359.

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

[71] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[72] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017.

[73] B. Liu, F. Li, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," in *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2023, pp. 1–5.

[74] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5918–5926.

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

[76] X. Zhao, Y. Wang, X. Cai, C. Liu, and L. Zhang, "Linear symmetric quantization of neural networks for low-precision integer hardware," in *International Conference on Learning Representations*, 2020.

[77] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[78] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Towards effective low-bitwidth convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7920–7928.

[79] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

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

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

[82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[83] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[84] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 4852–4861.

[85] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," in *International Conference on Learning Representations*, 2019.

[86] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

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

[89] S.-C. Zhou, Y.-Z. Wang, H. Wen, Q.-Y. He, and Y.-H. Zou, "Balanced quantization: An effective and efficient approach to quantized neural networks," *Journal of Computer Science and Technology*, vol. 32, no. 4, pp. 667–682, 2017.

[90] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[91] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[92] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[93] Y.-H. Liu and X.-J. Wang, "Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron," *Journal of computational neuroscience*, vol. 10, pp. 25–45, 2001.

[94] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952.

[95] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[96] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[97] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[98] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza *et al.*, "A low power, fully event-based gesture recognition system," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7243–7252.

[99] J.-K. Han, S.-Y. Yun, S.-W. Lee, J.-M. Yu, and Y.-K. Choi, "A review of artificial spiking neuron devices for neural processing and sensing," *Advanced Functional Materials*, vol. 32, no. 33, p. 2204102, 2022.

[100] S. Y. A. Yarga, J. Rouat, and S. Wood, "Efficient spike encoding algorithms for neuromorphic speech recognition," in *Proceedings of the International Conference on Neuromorphic Systems 2022*, 2022, pp. 1–8.

[101] Y. Zheng, Z. Yu, S. Wang, and T. Huang, "Spike-based motion estimation for object tracking through bio-inspired unsupervised learning," *IEEE Transactions on Image Processing*, vol. 32, pp. 335–349, 2022.

[102] B. Ruf and M. Schmitt, "Hebbian learning in networks of spiking neurons using temporal coding," in *International Work-Conference on Artificial Neural Networks*. Springer, 1997, pp. 380–389.

[103] A. Vigneron and J. Martinet, "A critical survey of stdp in spiking neural networks for pattern recognition," in *2020 international joint conference on neural networks (ijcnn)*. IEEE, 2020, pp. 1–9.

[104] F. Zenke and T. P. Vogels, "The remarkable robustness of surrogate gradient learning for instilling complex function in spiking neural networks," *Neural computation*, vol. 33, no. 4, pp. 899–925, 2021.

[105] N. Rathi, G. Srinivasan, P. Panda, and K. Roy, "Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation," in *International Conference on Learning Representations*, 2020.

[106] 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.

[107] B. Rueckauer, I.-A. Lungu, Y. Hu, and M. Pfeiffer, "Theory and tools for the conversion of analog to spiking convolutional neural networks," *arXiv preprint arXiv:1612.04052*, 2016.

[108] N.-D. Ho and I.-J. Chang, "Tcl: an ann-to-snn conversion with trainable clipping layers," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 793–798.

[109] B. Han, G. Srinivasan, and K. Roy, "Rmp-snn: Residual membrane potential neuron for enabling deeper high-accuracy and low-latency spiking neural network," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 13 558–13 567.

[110] S. Kim, S. Park, B. Na, and S. Yoon, "Spiking-yolo: spiking neural network for energy-efficient object detection," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 07, 2020, pp. 11 270–11 277.

[111] B. Han and K. Roy, "Deep spiking neural network: Energy efficiency through time based coding," in *European conference on computer vision*. Springer, 2020, pp. 388–404.

[112] H. Li, H. Liu, X. Ji, G. Li, and L. Shi, "Cifar10-dvs: an event-stream dataset for object classification," *Frontiers in neuroscience*, vol. 11, p. 309, 2017.

[113] P. Li, W. Qian, and D. Lilja, "A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic," in *IEEE 30th International Conference on Computer Design (ICCD)*, Sept 2012, pp. 303–308.

[114] J. Dickson, R. McLeod, and H. Card, "Stochastic arithmetic implementations of neural networks with in situ learning," in *IEEE Int. Conf. on Neural Networks*, 1993, pp. 711–716 vol.2.

[115] P.-S. Ting and J. Hayes, "Stochastic Logic Realization of Matrix Operations," in *17th Euromicro Conf. on Digital System Design (DSD)*, Aug 2014, pp. 356–364.

[116] G. Hinton, S. Osindero, and Y. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, July 2006.

# Appendices

# A

# Stochastic Computing

## A.1 Stochastic Computing and Its Computational Elements

In this appendix uppercase letters are used to denote elements of a stochastic stream, while lowercase letters represent the real values corresponding to those streams.

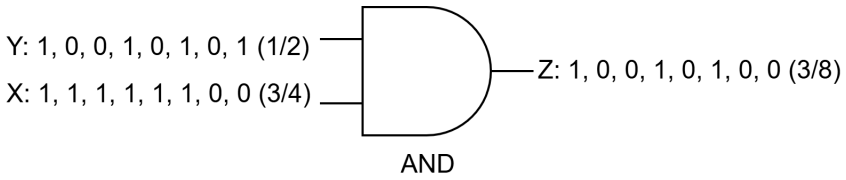
In stochastic computation, numbers are encoded as sequences of random bits. The value conveyed by the sequence is determined by the statistical properties of the bits rather than the individual bit values. Let $X \in \{0, 1\}$ represent a bit within the random sequence. To encode a real number $x \in [0, 1]$, the sequence is generated such that:

$$\mathbb{E}[X] = x, \tag{A.1}$$

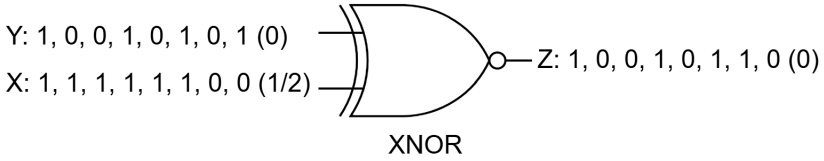
where $\mathbb{E}[X] = x$ represents the expected value of the random variable $X$. This representation is referred to as the *unipolar* format. To encode a signed real number $x \in [-1, 1]$, *bipolar* format is alternatively used:

$$\mathbb{E}[X] = (x + 1)/2. \tag{A.2}$$

Any real number can be expressed in one of these two formats by appropriately scaling it to




(a)




(b)

Figure A.1: Stochastic Multiplication in stochastic computing using (a) AND gate in unipolar format and (b) XNOR gate in bipolar format

fit within the required range. Additionally, a stochastic stream representing a real value $x$ is typically generated using a linear feedback shift register (LFSR) combined with a comparator. This unit is referred to as a binary-to-stochastic converter (B2S) [113].

### A.1.1 Multiplication In SC

In stochastic computation, the multiplication of two stochastic streams can be performed using AND and XNOR gates for the unipolar and bipolar encoding formats, respectively, as depicted in Figure A.1(a) and Figure A.1(b).

For the unipolar format, the multiplication of two input stochastic streams, $A$ and $B$, is calculated as:

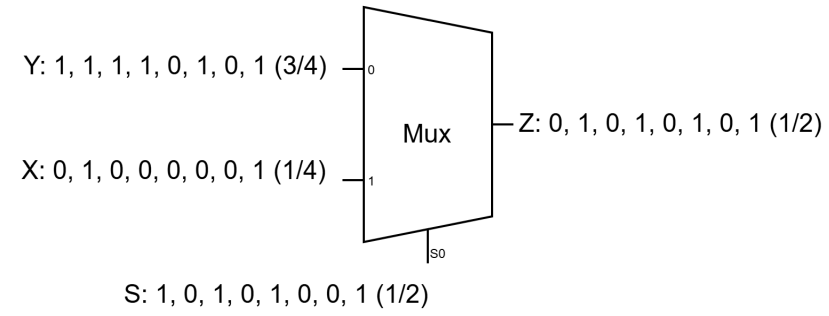
$$Y = \mathrm{AND}\,(A, B) = A \cdot B, \tag{A.3}$$

where "$\cdot$" represents the bit-wise AND operation. Assuming the input sequences are independent, the expected value of $Y$ is given by:

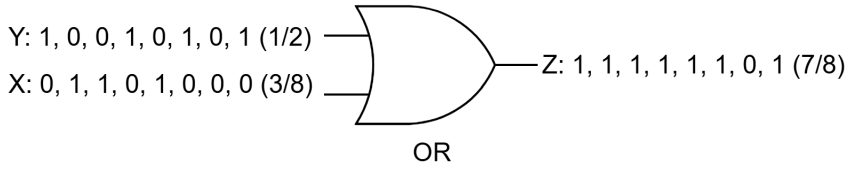
$$y = \mathbb{E}[Y] = a \times b. \tag{A.4}$$

where $a = \mathbb{E}[A]$ and $b = \mathbb{E}[B]$. For the bipolar format, multiplication is carried out as:

$$Y = \mathrm{XNOR}\,(A, B) = \mathrm{OR}\,(A \cdot B, (1 - A) \cdot (1 - B))\,, \tag{A.5}$$




(a)




(b)

Figure A.2: Addition in stochastic computing using (a) MUX and (b) OR gate

with the expected value:

$$\mathbb{E}[Y] = \mathbb{E}[A \cdot B] + \mathbb{E}[(1 - A) \cdot (1 - B)]. \tag{A.6}$$

Assuming independence of the input streams, this can be simplified as:

$$\mathbb{E}[Y] = \mathbb{E}[A] \times \mathbb{E}[B] + \mathbb{E}[1 - A] \times \mathbb{E}[1 - B]. \tag{A.7}$$

Further simplification yields:

$$y = 2\mathbb{E}[Y] - 1 = (2\mathbb{E}[A] - 1) \times (2\mathbb{E}[B] - 1)\,. \tag{A.8}$$

where $y$ is the real value corresponding to the resulting stochastic stream $Y$.

### A.1.2 Addition In SC

In stochastic computing, additions are typically performed using either scaled adders or OR gates [42, 114]. The scaled adder employs a multiplexer (MUX) for the addition operation.

The MUX output, $Y$ is given by:

$$Y = A \cdot S + B \cdot (1 - S)\,. \tag{A.9}$$

where $S$ is a stochastic stream acting as the select signal. If $S$ has a probability of 0.5, the expected value of $Y$ becomes $(\mathbb{E}[A] + \mathbb{E}[B])/2$, as illustrated in Figure A.2(a). This design ensures that the output remains within the legitimate range of the encoding format by scaling it down by a factor of 2. For $L$-input addition, a tree of multiple 2-input MUXes is used, resulting in an overall scaling factor of $L$. While this approach maintains correctness, the scaling can lead to precision loss, necessitating the use of longer bit-streams to achieve the desired accuracy. However, longer bit-streams increase latency.

Alternatively, OR gates can be employed as approximate adders, as shown in Figure A.2(b). The output $Y$ of an OR gate with inputs $A$, $B$ can be expressed as:

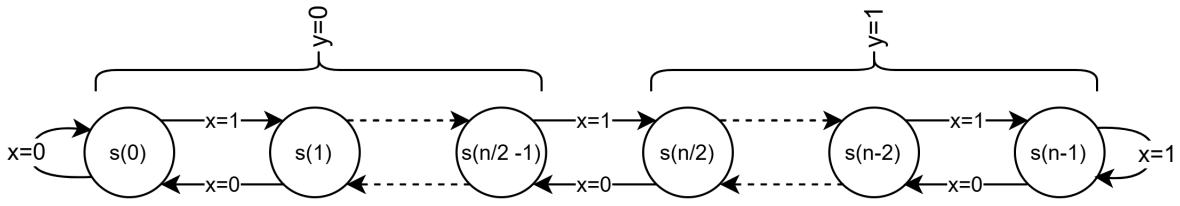
$$Y = A + B - A \cdot B. \tag{A.10}$$

OR gates function effectively as adders only when $\mathbb{E}[AB]$ is close to 0. To ensure this condition is satisfied, the inputs must first be scaled down. However, this scaling also reduces precision, requiring longer bit-streams to mitigate the loss, which similarly increases latency.

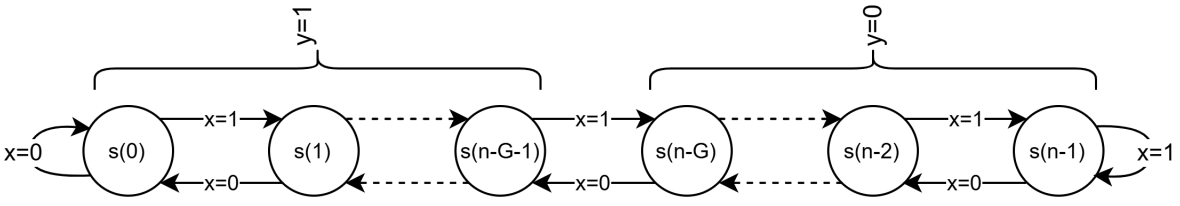
To address the precision loss and latency issues associated with scaled adders and OR gates, the Accumulative Parallel Counter (APC) was introduced in [115]. The APC operates by taking $N$ parallel bits as inputs and summing them in a counter at each clock cycle. This approach significantly reduces latency due to the small variance of the resulting sum. Unlike other methods, the APC converts the stochastic stream into binary format [115]. Consequently, its use is limited to scenarios where additions are either the final operation or require intermediate results in binary form.

### A.1.3 FSM-Based Functions In SC

Non-linear functions are fundamental components in both the inference and training processes of neural networks. During the training phase, the gradient of the non-linear function is essential for the backpropagation algorithm. A common practice is to employ an exact non-linear function with a well-defined and easily computable gradient [116]. Consequently, the same exact non-linear function, often implemented using a finite state machine (FSM) in the stochastic domain, is also utilized in the inference engine to perform classification tasks effectively. Among the most commonly used non-linear functions are the The hyperbolic




(a)




(b)

Figure A.3: State transition diagram of the FSM implementing (a) tanh and (b) exp functions

tangent (tanh) and exponentiation (exp). In the stochastic domain, these functions can be implemented using finite state machines (FSMs) [44]. Figure A.3(a) and Figure A.3(b) illustrate the state transition diagrams of FSMs designed for approximating tanh and exp functions, respectively.

The FSM for tanh is constructed to approximate the function as:

$$\tanh\left(\frac{nx}{2}\right) \approx 2 \times \mathbb{E}[\text{Stanh}\,(n, X)] - 1, \tag{A.11}$$

where $n$ represents the number of states in the FSM, $\mathbb{E}[\text{Stanh}\,(n, X)]$ and is the stochastic approximation of tanh. Both the input and output of the $\mathbb{E}[\text{Stanh}\,(n, X)]$ function are in bipolar format.

For the exponentiation function, the FSM is designed to approximate it as:

$$\exp\left(-2Gx\right) \approx \mathbb{E}[\text{Sexp}\,(n, G, X)] : x > 0. \tag{A.12}$$

where $G$ is the linear gain of the exponentiation function, $n$ is the number of FSM states, $\mathbb{E}[\text{Sexp}\,(n, G, X)]$ and is the stochastic approximation of exp. In this case, the input to the $\mathbb{E}[\text{Sexp}\,(n, G, X)]$ function is in bipolar format, while its output is in unipolar format.

### A.1.4 Integer Stochastic Stream

An integer stochastic stream is a sequence of integer values represented using either 2's complement or sign-and-magnitude formats [49]. The average value of this stream corresponds to a real number $s \in [0, m]$ for the unipolar format and $s \in [-m, m]$ for the bipolar format, where $m \in \{1, 2, \dots\}$. Essentially, the real value $s$ is the sum of probabilities from two or more binary stochastic streams. For example, the value 1.5 can be expressed as 0.625+0.875, with each probability represented as a conventional binary stochastic streams $\{1, 1, 1, 1, 1, 0, 0, 0\}$ and $\{1, 1, 1, 1, 1, 1, 1, 0\}$. Consequently, the integer stochastic representation of 1.5 is achieved by summing these binary stochastic streams, i.e., $\{2, 2, 2, 2, 2, 1, 1, 0\}$. More generally, an integer stochastic stream $S$ that represents a real value $s$ is a sequence composed of elements $S_i, i = \{1, 2, \dots, N\}$:

$$S_i = \sum_{j=1}^{m} X_i^j, \tag{A.13}$$

where $X_i^j$ represents each element of a binary stochastic sequence corresponding to a real value $x^j$. The expected value of the integer stochastic stream can then be expressed as:

$$s = \mathbb{E}[S_i] = \sum_{j=1}^{m} x^j. \tag{A.14}$$

Integer stochastic streams can also be represented in the bipolar format. In this case, the elements $S_i$ of the stream are defined as:

$$S_i = 2 \times \sum_{j=1}^{m} X_i^j - m, \tag{A.15}$$

The value represented by the stream is then given by:

$$s = \mathbb{E}[S_i] = 2 \times \sum_{j=1}^{m} \mathbb{E}[X_i^j] - m = 2 \times \sum_{j=1}^{m} x^j - m. \tag{A.16}$$

### A.1.5 FSM-Based Functions In Integral SC

The inputs to the stochastic FSM-based tanh adn exp functions are limited to real values within the [-1,1] interval. To achieve the desired Stanh or Sexp functions, the inputs must be scaled down, and the parameter$n$ in Equation (A.11) and Equation (A.12) must be adjusted.

**Algorithm 8:** Pseudo code of the conventional algorithm for FSM-based functions [49].

```
Data: Stochastic stream X_i ∈ {0,1} where i ∈ {1,2,...,N}
Result: Y_i
Counter ← Initial value;
for i ← 1 : N do
    Counter ← Counter + 2X_i - 1;
    if Counter > n-1 then
        Counter ← n-1;
    end
    if Counter < 0 then
        Counter ← 0;
    end
    if Counter > offset then
        Y_i ← 1;
    else
        Y_i ← 0;
    end
end
```

This adjustment typically increases the bit-stream length, resulting in higher latency. The transition between each state in the FSM is determined by the input value in bipolar format, which can either be 1 or 0. The state transitions are formulated in Algorithm 8 in conventional SC. In the algorithm, the bipolar input value is first mapped to either 1 or -1, depending on whether the input is 1 or 0, respectively. Then, the counter of the FSM is updated by adding the newly encoded values. These updates are analogous to the values in an integral stochastic stream with $m = 1$. Consequently, the conventional stochastic stream can be considered as representing the hard values of an integral stochastic stream. To extend this to integral SC, the FSM-based functions can be adapted to handle soft values. This involves modifying the conventional FSM-based functions to accommodate the continuous range of values in integral SC.

In integral SC, each element of a stochastic stream is represented using 2's complement or sign-magnitude representations within the range $\{-m, \ldots, m\}$ for the bipolar format. A state counter is adjusted by the integer input value $S_i \in \{-m, \ldots m\}$, where $i \in \{1, 2, ..., N\}$. This allows the state counter to be incremented or decremented by up to $m$ in each clock cycle, unlike conventional FSM-based functions, which are restricted to single-step transitions. The process for integer FSM-based functions is described in Algorithm 9. This algorithm

**Algorithm 9:** Pseudo code of the proposed algorithm for integer stochastic FSM-based functions [49].

**Data:** Integer value $S_i \in \{-m, ..., m\}$ where $i \in \{1, 2, ..., N\}$
**Result:** $Y_i$

```
Counter ← Initial value;
for i ← 1 : N do
    Counter ← Counter + S_i;
    if Counter > n × m-1 then
        Counter ← n × m-1;
    end
    if Counter < 0 then
        Counter ← 0;
    end
    if Counter > offset then
        Y_i ← 1;
    else
        Y_i ← 0;
    end
end
```

enables the integral SC framework to process values that may involve larger state transitions, offering increased flexibility compared to traditional stochastic FSMs, where only one-step transitions are typically allowed. The key advantage of this approach is its ability to handle a broader range of values and more complex state transitions, enabling more efficient and scalable implementations of stochastic functions.

The output of the integer FSM-based functions in the integral SC domain follows a similar encoding format as the conventional FSM-based functions. For example, the output of the integer Stanh function is in bipolar format, while the output of the integer exponentiation function is in unipolar format. Additionally, the integer FSM-based functions require $m$ times more states than their conventional counterparts. As a result, the approximate transfer functions of the integer Stanh and exp functions, referred to as IStanh and ISexp, respectively, are as follows:

$$\tanh\left(\frac{ns}{2}\right) \approx 2 \times \mathbb{E}[\text{IStanh}\,(m \times n, S)] - 1, \tag{A.17}$$

$$\exp\left(-2Gs\right) \approx \mathbb{E}[\text{ISexp}\,(m \times n, m \times G, S)] : s > 0. \tag{A.18}$$