Informatics

# Energy Efficiency and Fault Tolerance for Spiking and Deep Neural Networks

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktor der Technischen Wissenschaften**

by

**Rachmad Vidya Wicaksana Putra, M.Sc.**
Registration Number 11721413

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Muhammad Shafique

The dissertation has been reviewed by:

Assoc. Prof. Anupam Chattopadhyay

Prof. Dimitrios Soudris

Vienna, 17th July 2024

Rachmad Vidya Wicaksana Putra

# Declaration of Authorship

Rachmad Vidya Wicaksana Putra, M.Sc.

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 17th July 2024

Rachmad Vidya Wicaksana Putra

*The best of people are those who are most beneficial to people.*
— Prophet Muhammad PBUH —

*Knowledge is not merely what is memorized. Knowledge is what benefits.*
— Imam Shafi'i —

To my beloved parents, my parents-in-law, my wife: Mesa Dewi Puspita, my little kids: Raysa Rachmatillah, Ahsan Rachmatullah, and Zahra Rachmatillah.

# Acknowledgements

First of all, Alhamdulillah, all the praises and thanks belong to ALLAH for all of His blessings. Here, I would like to present my cordial gratitude to my advisor Prof. Dr. Muhammad Shafique for his invaluable supervision, inspiration, and motivation that helped me in improving my research skills. He guided me in exploring my research work and solving the challenging research issues comprehensively. His feedback, constructive criticism, and encouragement helped me in keeping my research momentum which resulted in the completion of this work.

I will present my special thanks to my colleagues: Muhammad Abdullah Hanif and Florian Kriebel. Abdullah has helped me with technical discussions on various research aspects. He has been very critical in scrutinizing my work. Without him, delivering high-quality work would be very difficult. Meanwhile, Florian has helped me in addressing multiple technical and non-technical issues, including my family reunification process. Without him, inviting my family to live in Vienna would have been very challenging.

I would like to present my thanks to all of my colleagues from the Computer Architecture and Robust Energy-Efficient Technologies (CARE-Tech) group for their feedback and discussions, i.e., (in alphabetical order) Alberto Marchisio, Bharath Srinivas Prabakaran, Le-Ha Hoang, Mahum Naseer, and Mahya Morid Ahmadi. I would also like to thank secretaries and technicians who have provided a good working environment.

Furthermore, I would also like to present my thanks to all of my colleagues from the TriLite Technologies GmbH. In particular, I want to thank Faraj Nassar, Vinicius Esteves Da Costa, Gregor Schestak, and Manuel Dorfmeister, who have guided and helped me through technical and non-technical discussions.

I am also grateful to the Indonesian Endowment Fund for Education (LPDP) Scholarship for financially supporting my research work, as well as to *Der Indonesische Islamische Verein in Austria* (WAPENA) and the Indonesian Student Association in Austria (PPI Austria) for providing good, productive, and vibrant environments for my family.

Last but not least, I would like to present my deepest gratitude to my parents (Purwinarti, Achmad Sumarsono, and Mustari), my parents-in-law (Mudji Lestari and Yulianto), my half sister (Rachmita Mustika Putri), and my brothers-in-law (Rangga Dwirian and Muhammad Wildan Sururi) for their sincere prayers, exceptional support, and sacrifices throughout my Ph.D. studies. Even though my late father has passed away when I was still a baby, his roles have been exceptionally fulfilled by my mother. She always reminds me to be grateful to ALLAH and be patient. Finally, I would also like to present my never-ending love to my wife (Mesa Dewi Puspita), and my amazing kids (Raysa Rachmatillah, Ahsan Rachmatullah, and Zahra Rachmatillah). They are always there praying for me, understanding my condition, and fueling my energies so that I can perform better and improve myself as a husband, a father, a son, a researcher, and a human being.

I dedicate this thesis to all of my family members and my advisor Prof. Dr. Muhammad Shafique.

# Abstract

Neural Networks (NNs) have become prominent Machine Learning (ML) algorithms as they achieve state-of-the-art accuracy for a wide range of data analytic applications, such as image classification, object recognition, healthcare systems, autonomous driving systems, and even business analytics. Therefore, deploying advanced NN algorithms, such as deep neural networks (DNNs) and spiking neural networks (SNNs), to resource- and energy-constrained embedded systems is an interesting research direction as it can enable many applications to improve the productivity of human life through better quality of services, higher efficiency, lower latency, as well as better security and privacy. Deploying NNs on embedded systems (i.e., so-called *embedded NN systems*) is a challenging task since NN algorithms are memory- and compute-intensive, thereby requiring large memory footprint and high energy consumption, which hinder the realization or limit the applicability of embedded NN systems. However, the existing solutions still face energy efficiency issues due to high memory access energy. Furthermore, the existing solutions also incur high memory and energy overheads for adapting to dynamically-changed environments, which make the offline-learned knowledge obsolete and degrade the accuracy at run time. Apart from energy efficiency issues, the existing solutions for SNN-based systems do not mitigate the negative impact of hardware-induced faults such as approximation errors (e.g., from reduced-voltage memories), permanent faults (e.g., from manufacturing defects), and transient faults (e.g., from the strikes of high energy particles). Therefore, alternate solutions are required to address the above challenges.

Toward this, the focus of our research is to provide a novel methodology that employs cross-layer hardware- and software-level techniques for improving energy efficiency and fault tolerance of spiking and deep neural networks. First, we improve DNN- and SNN-based systems by optimizing the off-chip memory (DRAM) access energy, as it dominates the total energy of NN-based systems. To do this, we optimize the number of DRAM accesses through the reduction of redundant accesses, and reduce the DRAM energy-per-access through effective data mapping in DRAM and novel DRAM architectures. Then, we enhance the NN-based systems to efficiently adapt to dynamic environments by improving the SNNs with a lightweight unsupervised continual learning mechanism. This learning mechanism is obtained through a learning algorithm that carefully learns new features while retaining the old yet important ones by leveraging the spiking activity. Furthermore, our research also improves the tolerance of SNN-based systems against hardware-induced faults. To do this, we mitigate approximation errors and permanent

faults by employing fault-aware training if the training dataset is fully available, and efficient fault-aware mapping without retraining if the training dataset is not fully available (e.g., due to IP and privacy reasons). Meanwhile, we mitigate transient faults (i.e., soft errors) by employing weight bounding and neuron protection techniques to minimize the negative impact of weight value changes and faulty neuron operations on accuracy. Furthermore, to support these fault mitigation techniques, we also propose lightweight hardware enhancements. All these techniques are integrated into our novel methodology to provide a judicious and synergistic design approach for enabling energy-efficient and fault-tolerant NN-based systems in diverse operating conditions/environments, which is crucial for resource- and energy-constrained embedded applications.

# Kurzfassung

**[Translation of the English version]**

*Neural Networks* (NNs) haben sich zu herausragenden Algorithmen des *Machine Learning* (ML) entwickelt, da sie für eine breite Palette von Datenanalyseanwendungen, wie Bildklassifizierung, Objekterkennung, Gesundheitssysteme, autonome Fahrsysteme und sogar Geschäftsanalysen, höchste Genauigkeit erreichen. Daher ist der Einsatz fortschrittlicher NN-Algorithmen wie *Deep Neural Networks* (DNNs) und *Spiking Neural Networks* (SNNs) für ressourcen- und energiebeschränkte eingebettete Systeme eine interessante Forschungsrichtung, da sie vielen Anwendungen ermöglichen kann, die Produktivität des menschlichen Lebens durch bessere Servicequalität, höhere Effizienz, geringere Latenz sowie bessere Sicherheit und Privatsphäre zu verbessern. Der Einsatz von NNs auf eingebetteten Systemen (d. h. so genannten *eingebetteten NN-Systemen*) ist eine anspruchsvolle Aufgabe, da NN-Algorithmen speicher- und rechenintensiv sind und daher einen großen Speicherbedarf und einen hohen Energieverbrauch erfordern, was die Realisierung eingebetteter NN-Systeme behindert oder ihre Anwendbarkeit einschränkt. Die vorhandenen Lösungen haben jedoch aufgrund des hohen Energieverbrauchs beim Speicherzugriff immer noch mit Energieeffizienzproblemen zu kämpfen. Darüber hinaus verursachen die bestehenden Lösungen auch einen hohen Speicher- und Energieaufwand für die Anpassung an dynamisch veränderte Umgebungen, wodurch das offline erlernte Wissen obsolet wird und die Genauigkeit zur Laufzeit abnimmt. Abgesehen von Problemen mit der Energieeffizienz mildern die bestehenden Lösungen für SNN-basierte Systeme nicht die negativen Auswirkungen von hardwarebedingten Fehlern wie Approximationsfehlern (z. B. durch Speicher mit reduzierter Spannung), permanenten Fehlern (z. B. durch Fertigungsfehler) und transienten Fehlern (z. B. durch Einschläge hochenergetischer Partikel). Daher sind alternative Lösungen erforderlich, um die oben genannten Herausforderungen zu bewältigen.

In diesem Zusammenhang liegt der Schwerpunkt unserer Forschung auf der Bereitstellung einer neuartigen Methodik, die schichtübergreifende Techniken auf Hardware- und Softwareebene zur Verbesserung der Energieeffizienz und Fehlertoleranz von *Spiking*- und *Deep Neural Networks* verwendet. Zunächst verbessern wir DNN- und SNN-basierte Systeme, indem wir die Zugriffsenergie auf den Off-Chip-Speicher (DRAM) optimieren, da diese die Gesamtenergie von NN-basierten Systemen dominiert. Zu diesem Zweck optimieren wir die Anzahl der DRAM-Zugriffe durch Reduzierung redundanter Zugriffe und reduzieren den DRAM-Energieverbrauch pro Zugriff durch effektives Datenmapping in DRAM und neuartigen DRAM-Architekturen. Anschließend verbessern wir die NN-basierten Systeme, um sie effizient an dynamische Umgebungen anzupassen, indem wir die SNNs mit einem leichten, unbeaufsichtigten kontinuierlichen Lernmechanismus verbessern. Dieser Lernmechanismus wird durch einen Lernalgorithmus erreicht, der sorgfältig neue Funktionen lernt und gleichzeitig die alten, aber wichtigen beibehält, indem er die Spiking-Aktivität nutzt. Darüber hinaus verbessert unsere Forschung auch die Toleranz von SNN-basierten Systemen gegenüber hardwarebedingten Fehlern. Zu diesem Zweck mildern wir Approximationsfehlern und permanente Fehler, indem wir fehlerbewusstes Training einsetzen, wenn der Trainingsdatensatz vollständig verfügbar ist, und effizientes fehlerbewusstes Mapping ohne erneutes Training, wenn der Trainingsdatensatz nicht vollständig verfügbar ist (z. B. aus IP- und Datenschutzgründen). Gleichzeitig mildern wir transienten Fehler (d. h. *Soft Errors*) durch den Einsatz von Gewichtsbegrenzungs- und Neuronenschutztechniken, um die negativen Auswirkungen von Gewichtswertänderungen und fehlerhaften Neuronenoperationen auf die Genauigkeit zu minimieren. Darüber hinaus schlagen wir zur Unterstützung dieser Fehlerminderungstechniken auch leichte Hardwareverbesserungen vor. Alle diese Techniken sind in unsere neuartige Methodik integriert, um einen umsichtigen und synergetischen Designansatz für die Ermöglichung energieeffizienter und fehlertoleranter NN-basierter Systeme unter unterschiedlichen Betriebsbedingungen/-umgebungen bereitzustellen, was für ressourcen- und energiebeschränkte eingebettete Anwendungen von entscheidender Bedeutung ist.

# Publications of this PhD Thesis

## Journals

[1] R. V. W. Putra and M. Shafique, "FSpiNN: An Optimization Framework for Memory-Efficient and Energy-Efficient Spiking Neural Networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, vol. 39, no. 11, pp. 3601–3613, 2020.

[2] R. V. W. Putra, M. A. Hanif, and M. Shafique, "ROMANet: Fine-Grained Reuse-Driven Off-Chip Memory Access Management and Data Organization for Deep Neural Network Accelerators," *IEEE Transactions on Very Large Scale Integration Systems (**TVLSI**)*, pp. 1–14, 2021.

[3] R. V. W. Putra, M. A. Hanif, and M. Shafique, "EnforceSNN: Enabling Resilient and Energy-Efficient Spiking Neural Network Inference considering Approximate DRAMs for Embedded Systems," *Frontiers in Neuroscience (**FNINS**)*, vol. 16, 2022.

[4] R. V. W. Putra, M. A. Hanif, and M. Shafique, "RescueSNN: Enabling Reliable Executions on Spiking Neural Network Accelerators under Permanent Faults," *Frontiers in Neuroscience (**FNINS**)*, vol. 17, 2023.

[5] R. V. W. Putra, M. A. Hanif, and M. Shafique, "PENDRAM: High-Performance and Energy-Efficient Deep Neural Networks using a Generalized DRAM Data Mapping Policy," *Submitted to IEEE Journal*.

## Conferences

[1] R. V. W. Putra, M. A. Hanif and M. Shafique, "SoftSNN: Low-Cost Fault Tolerance for Spiking Neural Network Accelerators under Soft Errors", *in the 59th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 151-156, July 2022. ***Received a HIPEAC Paper Award & Selected for ACM Showcase***

[2] R. V. W. Putra, M. A. Hanif, and M. Shafique, "SparkXD: A Framework for Resilient and Energy-Efficient Spiking Neural Network Inference using Approximate

DRAM", *in the 58th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 379-384, December 2021. ***Received a HIPEAC Paper Award***

[3] R. V. W. Putra and M. Shafique, "SpikeDyn: A Framework for Energy-Efficient Spiking Neural Networks with Continual and Unsupervised Learning Capabilities in Dynamic Environments", *in the 58th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 1057-1062, December 2021. ***Received a HIPEAC Paper Award***

[4] R. V. W. Putra, M. A. Hanif, and M. Shafique, "DRMap: A generic dram data mapping policy for energy-efficient processing of convolutional neural networks", *in the 57th ACM/IEEE Design Automation Conference (**DAC**)*, Virtual Event, pp. 1–6, July 2020. ***Received a HIPEAC Paper Award***

[5] R. V. W. Putra, M. A. Hanif and M. Shafique, "ReSpawn: Energy-Efficient Fault-Tolerance for Spiking Neural Networks considering Unreliable Memories", *in the IEEE/ACM International Conference on Computer Aided Design (**ICCAD**)*, Munich, Germany, pp. 1-9, November 2021.

[6] R. V. W. Putra and M. Shafique, "Q-SpiNN: A Framework for Quantizing Spiking Neural Networks", *in the International Joint Conference on Neural Networks (**IJCNN**)*, Virtual Event, pp. 1-8, July 2021.

[7] R. V. W. Putra and M. Shafique, "lpSpikeCon: Enabling Low-Precision Spiking Neural Network Processing for Efficient Unsupervised Continual Learning on Autonomous Agents", *in the International Joint Conference on Neural Networks (**IJCNN**)*, Padova, Italy, pp. 1-8, July 2022.

[8] R. V. W. Putra and M. Shafique, "Mantis: Enabling Energy-Efficient Autonomous Mobile Agents with Spiking Neural Networks", *in the 9th International Conference on Automation, Robotics and Applications (**ICARA**)*, Abu Dhabi, UAE, pp. 1-5, February 2023.

In these publications, as the first author, I contributed to the problem definition, as well as to the concept, developments, implementations, and validations of the proposed solutions. Meanwhile, my co-authors including my advisor contributed through extensive discussions on the problem definition, solution developments, and results analyses.

# Other Co-Authored Publications

## Book Chapters

[1] M. A. Hanif, F. Khalid, R. V. W. Putra, M. T. Teimoori, F. Kriebel, J. Zhang, K. Liu, S. Rehman, T. Theocharides, A. Artusi, S. Garg, M. Shafique, "Robust Computing for Machine Learning-Based Systems", in *Dependable Embedded Computing*, Springer Science+Business Media, LLC, pages 479-503, 2019.

## Journals

[1] P. Achararit, M. A. Hanif, R. V. W. Putra, M. Shafique, and Y. Hara-Azumi, "APNAS: Accuracy-and-Performance-Aware Neural Architecture Search for Neural Hardware Accelerators", *IEEE Access*, vol. 8, pp. 165319-165334, 2020.

## Conferences/Workshops

[1] R. V. W. Putra, and M. Shafique, "TopSpark: A Timestep Optimization Methodology for Energy-Efficient Spiking Neural Networks on Autonomous Mobile Agents," *in the IEEE/RSJ International Conference on Intelligent Robots and Systems (**IROS**)*, Detroit, Michigan, USA, pp. 1-5, October 2023.

[2] M. Shafique, A. Marchisio, R. V. W. Putra, and M. A. Hanif, "Towards Energy-Efficient and Secure Edge AI: A Cross-Layer Framework," *in the IEEE/ACM International Conference on Computer-Aided Design (**ICCAD**)*, Munich, Germany, pp. 1-9, November 2021.

[3] M. A. Hanif, F. Khalid, R. V. W. Putra, S. Rehman, and M. Shafique, "Robust Machine Learning Systems: Reliability and Security for Deep Neural Networks", *in the 24th International Symposium on On-Line Testing And Robust System Design (**IOLTS**)*, Platja d'Aro, Spain, pp. 257-260, July 2018.

[4] A. Marchisio, R. V. W. Putra, M. A. Hanif, and M. Shafique, "HW/SW Co-Design and Co-Optimizations for Deep Learning," *in the Embedded Systems Week (**ESWEEK**) Workshop on Intelligent Embedded Systems Architectures and Applications (**INTESA**)*, Torino, Italy, pp. 13–18, 2018

# Awards and Achievements

[1] Multiple **HiPEAC Paper Awards** for first-authored papers at the Design Automation Conference (DAC).

- R. V. W. Putra, M. A. Hanif and M. Shafique, “SoftSNN: Low-Cost Fault Tolerance for Spiking Neural Network Accelerators under Soft Errors”, *in the 59th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 151-156, July 2022.
- R. V. W. Putra, M. A. Hanif, and M. Shafique, “SparkXD: A Framework for Resilient and Energy-Efficient Spiking Neural Network Inference using Approximate DRAM”, *in the 58th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 379-384, December 2021.
- R. V. W. Putra and M. Shafique, “SpikeDyn: A Framework for Energy-Efficient Spiking Neural Networks with Continual and Unsupervised Learning Capabilities in Dynamic Environments”, *in the 58th ACM/IEEE Design Automation Conference (**DAC**)*, San Francisco, USA, pp. 1057-1062, December 2021.
- R. V. W. Putra, M. A. Hanif, and M. Shafique, “DRMap: A generic dram data mapping policy for energy-efficient processing of convolutional neural networks”, *in the 57th ACM/IEEE Design Automation Conference (**DAC**)*, Virtual Event, pp. 1–6, July 2020.

[2] Our SoftSNN paper from the 59th Design Automation Conference 2022 got selected for the **ACM Showcase**.

# Contents

CHAPTER 1

# Introduction

*Artificial Intelligence* (AI) has been widely used for organizing, analyzing, and inferring information from digital data. The reason is that *Machine Learning* (ML)-based AI has demonstrated state-of-the-art performance in many data analytic tasks [LBH15], such as object recognition [KSH12, SZ14, HZC$^+$17], smart home [MPC16, ZAJ$^+$19], smart healthcare [M$^+$18, PAR$^+$21], and smart automotive [ABAR17, MAD$^+$20]. The state-of-the-art performance of ML algorithms is achieved through *Brain-inspired Computations* that take inspiration from how a human brain works, so-called *Neural Networks* (NNs). The computation models in advanced NNs can be categorized into two approaches, i.e., *Deep Neural Networks* (DNNs) and *Spiking Neural Networks* (SNNs). An overview of the relation between AI, ML, and NNs (DNNs and SNNs) is shown in Figure 1.1. These advanced NN algorithms can improve the productivity of users, hence the deployment of such algorithms for many AI applications, especially the ones with resource- and energy-constrained computing platforms (e.g., embedded systems), is highly desired.

## 1.1 Research Motivation

### 1.1.1 Trends of Application Use-Cases of Neural Networks

The widespread use of powerful computing platforms, such as CPUs, (general purpose or embedded) GPUs, or specialized hardware accelerators in servers, personal computers, or smartphones, has enabled the deployment of NN algorithms for diverse ML applications, such as smart healthcare, smart robots/agents, smart home, smart transportation, smart factory/industry, smart automotive, smart grid, and smart assistant [LBH15, SMWPH21]; see Figure 1.2. Recently, researchers have started studies on how to bring the remarkable capabilities of NNs to resource- and energy-constrained embedded systems (such as edge computing). This is to improve the productivity of users through better quality of services, lower latency, higher energy efficiency, as well as better security and privacy,

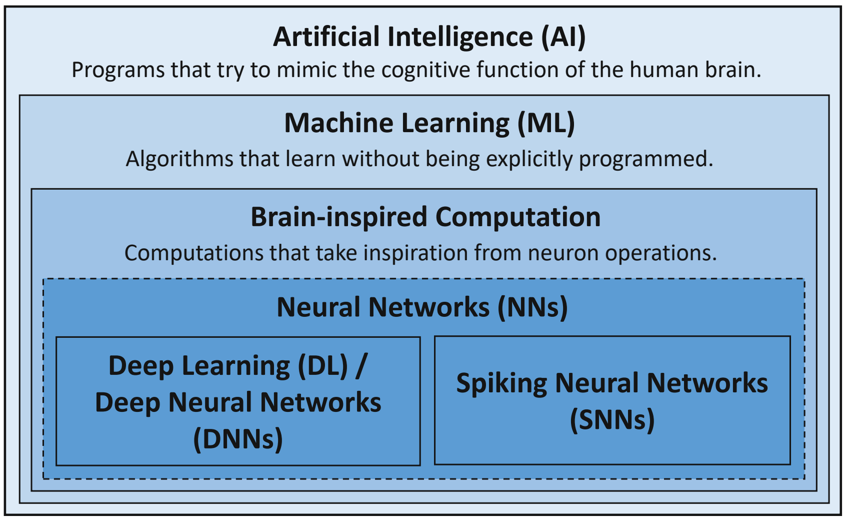

Artificial Intelligence (AI)
Programs that try to mimic the cognitive function of the human brain.
Machine Learning (ML)
Algorithms that learn without being explicitly programmed.
Brain-inspired Computation
Computations that take inspiration from neuron operations.
Neural Networks (NNs)
Deep Learning (DL) / Deep Neural Networks (DNNs)
Spiking Neural Networks (SNNs)

Figure 1.1: The relation of AI, ML, and NNs which encompass DNNs and SNNs. AI is defined as the science and engineering field that develops intelligent machines with cognitive capabilities like humans, while ML refers to concepts/algorithms that enable computers to learn without being explicitly programmed [SCYE17].

thereby enabling diverse energy-efficient ML systems, such as ML-powered smartphones and IoT-Edge devices. These studies are considered even more important nowadays since the number of smartphones and IoT-connected devices has increased significantly in the last few years and is estimated to stay increasing in the future, as shown in Figure 1.3. Therefore, *the successful deployment of NNs on embedded systems will enable more application use-cases with stringent memory and energy constraints to benefit from NN algorithms.* To accomplish this, understanding the requirements of NNs is important, which will be discussed in Section 1.1.2.

### 1.1.2 Requirements of Neural Networks

There are two prominent NN paradigms that are actively studied and explored, i.e., DNNs and SNNs, each having unique computational models [SNT$^+$20a]. DNNs employ the weighted sum neuron model, and supervised learning schemes like gradient descent-based backpropagation. DNNs are now used extensively in practical ML systems, as they achieve state-of-the-art accuracy [HMD16]. Trends in DNN developments show that higher accuracy can be achieved through larger model sizes and higher complexity of computations, as illustrated in Figure 1.4. Meanwhile, SNNs employ biological neuron models like Leaky Integrate-and-Fire (LIF), and biological learning schemes like Spike-Timing-Dependent Plasticity (STDP), which can be performed in an unsupervised manner. Therefore, SNNs bear the potential for having lower energy consumption than DNNs due to their sparse spike-based computations. Trends in SNN developments also show that higher accuracy can be achieved through larger model sizes and higher complexity of computations, as illustrated in Figure 1.5.

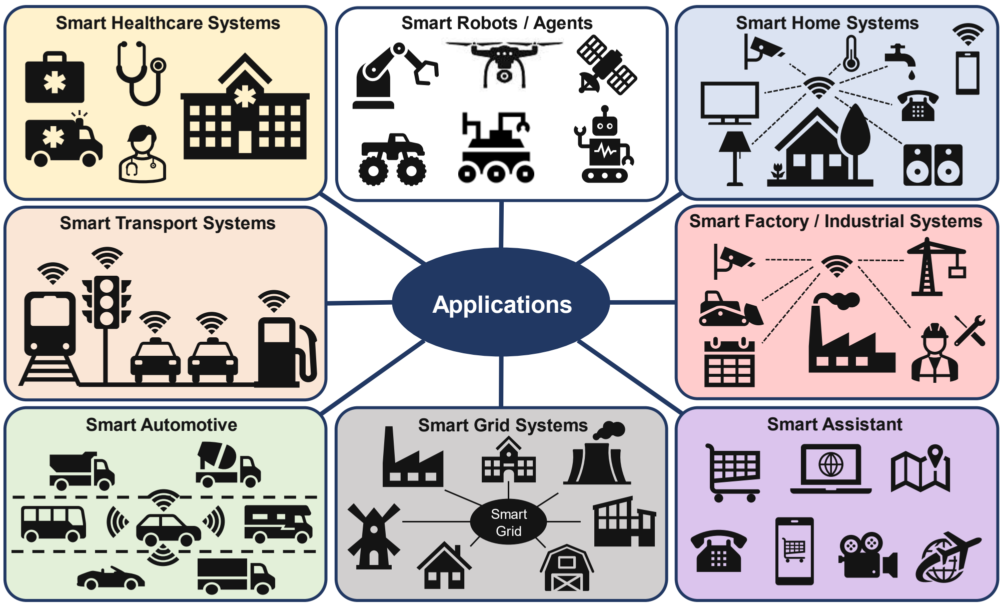


Figure 1.2: Prominent applications of NN-based ML algorithms.

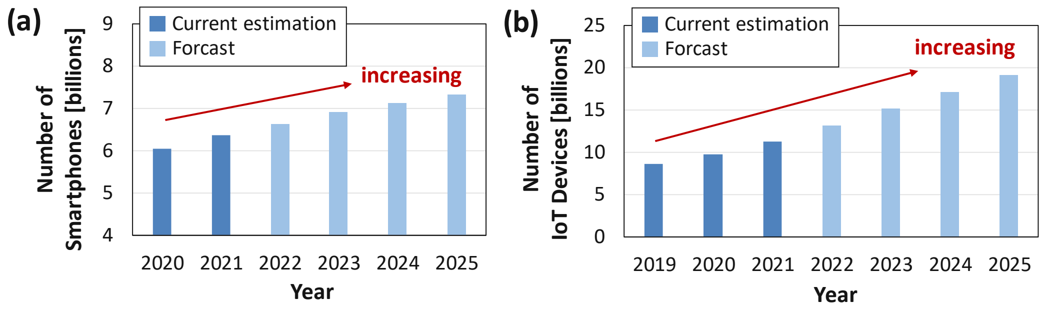


Figure 1.3: (a) Current estimation and forecast of the number of smartphones, whose data are obtained from [ban]. (b) Current estimation and forecast of the number of IoT-connected devices, whose data are obtained from [Vai].

Consequently, executing high-accuracy NN models (DNNs and SNNs) will impose high-complexity computations and high energy consumption, thereby making it challenging to implement them in resource-constrained computing platforms [SMWPH21]. Various optimization techniques (e.g., pruning and quantization) and specialized hardware accelerators have been proposed to expedite the NN processing. However, these NN-based systems still face energy efficiency challenges due to the DRAM-based off-chip memory energy, which is higher than the energy for other operations and dominates the systems' total energy [SCYE17, CBM$^+$20a, CBM$^+$20b]. Therefore, *minimizing the DRAM access energy is the key to substantially improving the overall energy efficiency of both the DNN and SNN systems.* Furthermore, real-world environments may have diverse operational conditions, which may not be completely accommodated at the design time of ML systems. Hence, the deployed systems should have the capability to learn new

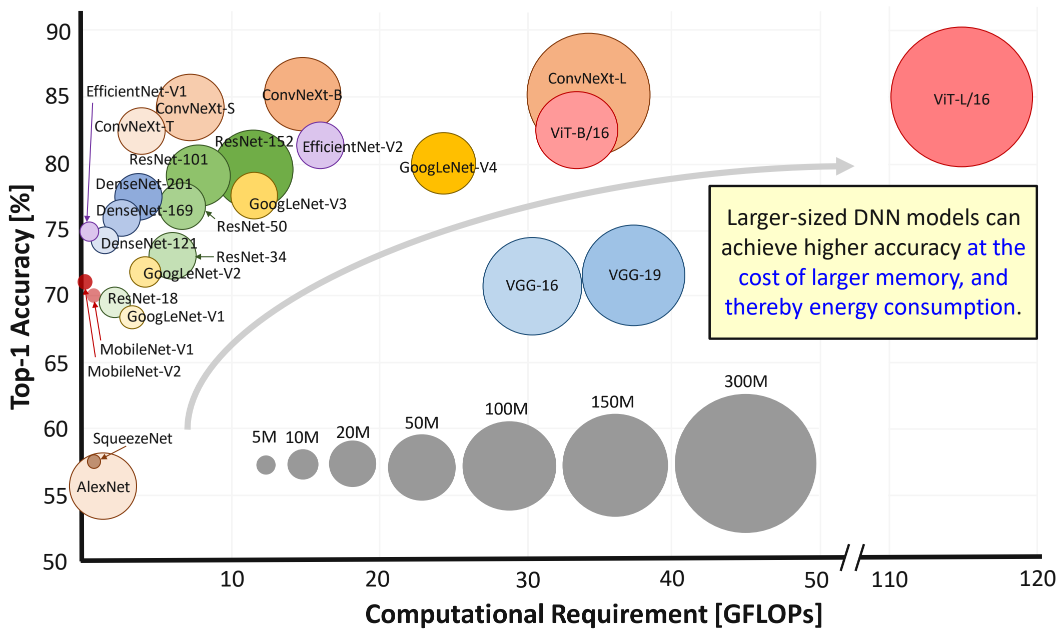


Figure 1.4: Characteristics of DNN models for image classification task on the ImageNet dataset considering the accuracy scores, model sizes, and computational requirements. Data are obtained from [Ope, pap, mmc].

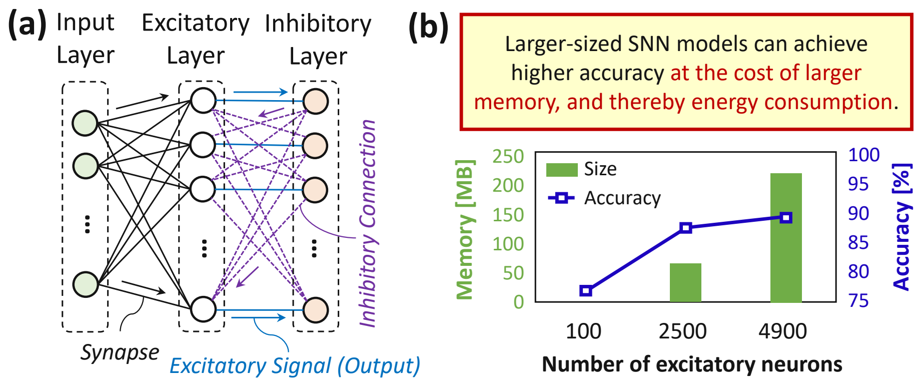

(a)
Input Layer
Excitatory Layer
Inhibitory Layer
Inhibitory Connection
Synapse
Excitatory Signal (Output)
(b)
Larger-sized SNN models can achieve higher accuracy at the cost of larger memory, and thereby energy consumption.
Memory [MB]
Accuracy [%]
Size
Accuracy
Number of excitatory neurons
100
2500
4900

Figure 1.5: (a) SNN architecture that supports bio-plausible learning rule (e.g., STDP) and unsupervised learning settings. (b) Characteristics of SNN models for image classification task on the MNIST dataset considering the accuracy scores and model sizes [PS20].

data/features online for adapting to dynamic environments, since the information learned offline can be obsolete or may lead to low accuracy at run-time under changing scenarios [PARR18]. Moreover, new data that are gathered directly from environments are usually unlabeled [RPR19], and their classes might not be randomly distributed, thereby making the systems difficult to learn different classes/tasks proportionally [AR16, AR20]. *For this purpose, we focus on the SNN systems as they support unsupervised learning capabilities which are required for addressing the given challenge.*

These NN-based systems are also expected to produce reliable output under the presence of hardware-induced reliability threats, such as approximation errors from approximate hardware units (e.g., memories), permanent faults from manufacturing defects, and transient faults from strikes of high-energy particles. It is important since the hardware-induced faults can lead to accuracy degradation, and decrease the yield of wafer chips. *For this purpose, we focus on the SNN systems as their fault tolerance under hardware-induced faults has not been extensively explored yet.*

In summary, *NN-based systems should have high energy efficiency and high tolerance against hardware-induced faults to enable their reliable embedded implementation.*

## 1.2 Research Problems and Challenges

Section 1.1.2 highlights that energy efficiency and fault tolerance aspects are the key requirements for the successful deployment of NNs on embedded systems. Therefore, *the targeted research problems are about how to achieve energy-efficient and fault-tolerant NN-based systems*; see the overview in Figure 1.6. Solving these problems impose scientific research challenges, as highlighted in the following.

- **Challenges for Energy Efficiency Techniques:** In DNN systems, most of previous works [ZLS+15, ZSF+19, LYL+18] only presented the isolated (on-chip) DNN accelerator design, and did not thoroughly study the impact of DRAM accesses, especially when the full DNN processing cannot be mapped on an accelerator fabric at the same time. Therefore, *the related challenge is to understand the impact of DRAM accesses considering the dataflow of DNN processing, and then leverage this information for saving DRAM access energy.* Meanwhile, in SNN systems, the existing works still focus on improving accuracy but at the cost of a huge amount of additional computations [S+17, HSS+18, SPH+19, HSS+19], thereby leading to a high memory footprint and energy consumption. Other SNN works proposed various bio-plausible learning techniques to achieve high accuracy in dynamically changed environments, but at the cost of additional operations [AR16, AR20], non-proportional quantities of training samples [PARR18], and large memory footprint and complex exponential calculations [PARR18]. Therefore, *the related challenge is to optimize SNNs for reducing memory footprint and energy consumption, while devising a lightweight unsupervised learning mechanism considering both dynamic and non-dynamic environments.*

- **Challenges for Fault Tolerance Techniques:** The existing works still focus on the software-level fault modeling for SNNs [VDNA19, SPMJ+20]. Therefore, the impact of hardware-induced faults (e.g., approximation errors, permanent faults, and transient faults) in system-level accuracy considering SNN hardware architectures, and the respective fault mitigation techniques, are still an unexplored avenue. Therefore, *the related challenge is to devise cost-effective fault tolerance for SNNs against approximation errors, permanent faults, and transient faults.*

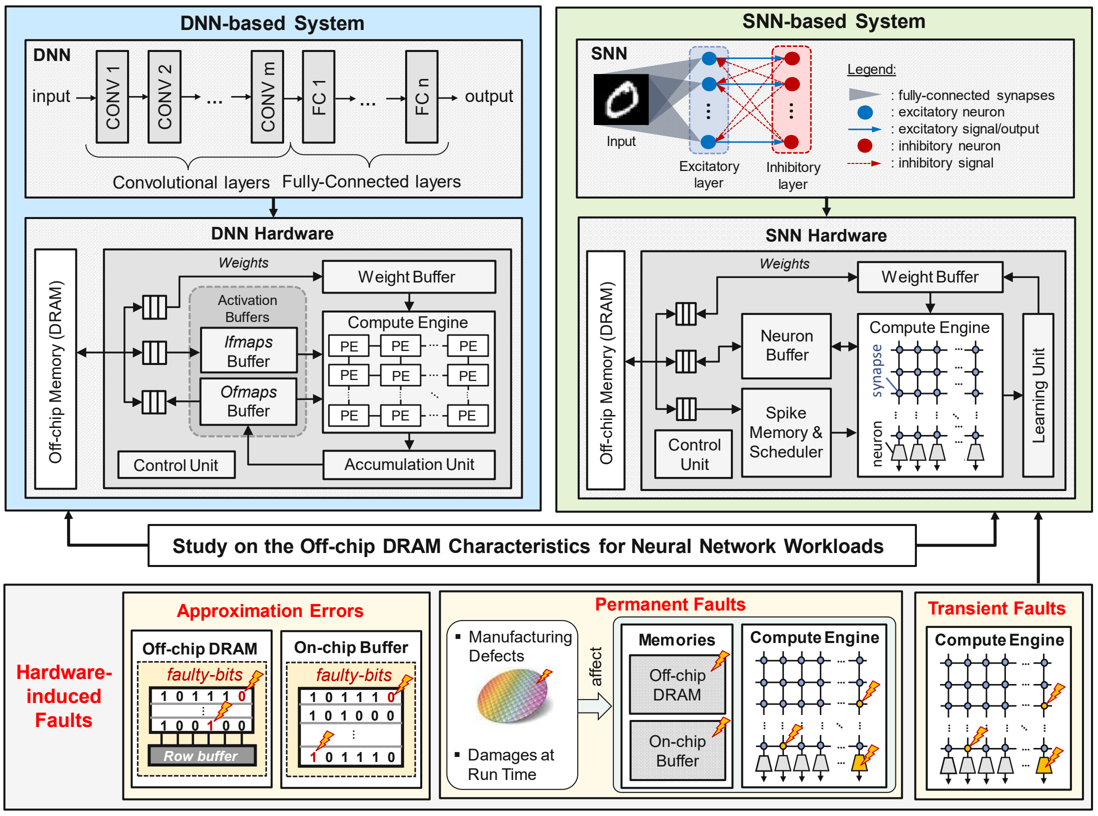


Figure 1.6: Overview of the targeted problems in this research. For the DNN-based system, our works will investigate techniques for improving energy efficiency. Meanwhile, for the SNN-based system, our works will investigate techniques to improve energy efficiency in both non-dynamic and dynamic environments, as well as resilience against hardware-induced errors.

## 1.3 Summary of the State-of-the-Art Techniques and Their Limitations

In this section, we present the summary of state-of-the-art works for improving the energy efficiency and fault tolerance of NN-based systems, and their limitations. In specific, for DNN systems, the presented works are related to techniques for optimizing energy consumption. For SNN systems, the presented works are related to techniques for optimizing energy consumption and improving resilience against hardware-induced faults.

### 1.3.1 Energy-Efficient DNN Systems

Some works employed pruning [HMD16, AHS17, HLL+18] and quantization [GAGN15, HMD16, JKC+18a] to compress the DNN model size for reducing the number of DRAM accesses, and thereby the systems' total energy. Other works employed data partitioning and scheduling to minimally move the data from DRAM to on-chip memory [ZLS$^+$15, ZSF$^+$19, LYL$^+$18, TKP20]. However, there are several limitations to the state-of-the-art works as discussed in the following.

1. *DRAM access optimization:* The existing works do not minimize redundant DRAM accesses for overlapping data in convolutional operations. Therefore, their estimation regarding the number of DRAM accesses provides sub-optimal results.

2. *DRAM energy-per-access optimization:* The existing works do not optimize the DRAM energy-per-access, which varies depending on the internal state of DRAM, hence leading to sub-optimal DRAM access energy and latency.

3. *Data mapping policy for the on-chip memory:* The existing works do not devise a data mapping policy for the on-chip memory that efficiently moves data between the DRAM and the compute engine, hence leading to sub-optimal on-chip access energy and throughput.

### 1.3.2 Energy-Efficient SNN Systems

Some works employed pruning [RPR19, EBDB20, GFY$^+$20] and quantization [RPR19] to compress the SNN model size for reducing the memory access requirements. Other works employed approximate operations [SVR17] and data bundling to reduce memory accesses [KSVR19]. Hardware accelerators have also been employed to improve the efficiency of SNN processing, but they do not exploit the intrinsic resilience characteristics of SNNs, thereby limiting further energy savings. Meanwhile, to achieve learning capabilities in dynamic environments, existing works proposed to employ a set of reserved synapses and neurons for learning new features [AR16], a weight decay to remove the learned information and provide spaces for learning new features [PARR18], and a set of specialized neurons [AR20]. However, there are several limitations to the state-of-the-art works as discussed in the following.

1. *SNN model optimization:* SNN systems with unsupervised learning capabilities typically employ a pair of excitatory and inhibitory neurons to support the learning process. However, the existing works do not optimize the functionality of inhibitory neurons, thereby incurring large memory and energy consumption.

2. *Hardware-level optimization:* The existing works do not consider hardware-level optimization (e.g., employment of approximate hardware) that can substantially reduce the operational power/energy of the SNN systems, while exploiting the resilience nature of SNNs.

3. *Improvement of the learning mechanism:* The existing works suffer from spurious weight updates during training as the weight update happens at every spike, thereby leading to accuracy degradation. For dynamic environments, they consider learning mechanisms that require additional components (e.g., neurons), complex exponential calculations, and/or non-proportional quantities of training samples, thereby incurring high memory and energy consumption.

### 1.3.3 Fault-Tolerant SNN Systems

State-of-the-art works on fault tolerance for SNNs still focus on modeling possible types of faults [VDNA19] and studying the impact of a specific fault (e.g., bit flips or synapses removal) on accuracy [SPMJ+20, VMA+20, RLIS21], without considering the underlying hardware architecture. Therefore, *the impact of hardware-induced faults in system-level accuracy considering SNN neuromorphic architectures, and the respective fault mitigation techniques, are still an unexplored avenue.*

## 1.4 Scientific Research Objectives

To systematically address the targeted research problems, while considering the associated research challenges and limitations of state-of-the-art works, we define the following major scientific objectives for the thesis.

1. **Optimizing the memory access energy for DNN systems:** Most of the state-of-the-art works observed that optimizing the data partitioning and scheduling is effective to reduce the DRAM accesses and improve the energy of DNN systems. However, they do not optimize the redundant DRAM accesses for the overlapping data partition in feature maps, and the DRAM energy-per-access.

   *Objective: We aim to optimize the redundant DRAM accesses for the overlapping data partition in feature maps, and the DRAM energy-per-access. Besides DRAM access optimization, we also investigate the data mapping strategy for the on-chip memory to efficiently shuttle data between the DRAM and the compute engine. Furthermore, we investigate how to exploit the new memory architectures from the literature, such as DRAM with subarray-level parallelism (SALP) and tiered-latency DRAM (TL-DRAM), to further optimize the DRAM access energy, and thereby the DNN systems' energy.*

2. **Optimizing the memory access energy for SNN systems:** The state-of-the-art works employed pruning, quantization, and data bundling to optimize the memory access energy. However, these techniques incur high overheads for data encoding and may suffer from accuracy degradation due to information loss. Moreover, they do not optimize the inhibitory operations which incur considerable memory and energy consumption.

   *Objective: We aim to minimize the memory requirement of SNNs by optimizing the inhibitory operations and employing quantization. To deal with the loss of inhibitory operations, a cost-effective compensation technique is also investigated. In this manner, memory footprint is expected to be reduced significantly, without losing accuracy. This approach keeps all excitatory neurons and synapses active, so that they can be maximally used for learning new features at run-time, which is required for unsupervised continual learning settings.*

3. **Investigating the effective hardware-level optimization for SNN systems:** In hardware-level optimization, state-of-the-art works employed hardware accelerators to improve the performance efficiency of SNN processing. However, such solutions consume high design time and do not consider exploiting the resilience characteristics of SNNs through hardware-level approximations, which have the potential to substantially reduce the processing power/energy.

   *Objective: We focus on hardware-level optimization for SNN systems by employing approximate hardware on SNN hardware accelerators, e.g., off-chip and on-chip memories. To find the appropriate approximation configurations, we design and employ design space exploration. In this manner, this approach will judiciously reduce the processing power/energy of the SNN systems, while meeting the design constraints (e.g., accuracy).*

4. **Investigating the unsupervised continual learning for SNN systems:** The state-of-the-art techniques suffer from spurious weight updates, as their weight update happens at each spike. Moreover, they incur high memory requirements and processing energy due to their large resources (e.g., inhibitory and additional neurons) and complex exponential computations, thereby hindering the SNN systems to perform unsupervised continual learning at run-time in tight resource budgets.

   *Objective: First, we aim to experimentally study the impact of the inhibitory layer and different SNN parameters (e.g., weight decay and neurons' membrane threshold potential) on accuracy, under non-dynamic and dynamic environments. These observations will provide information that is required to optimize and control the neuron behavior and spiking activity, which are important for bio-plausible unsupervised learning. Then, we hope to leverage this information to determine how the learning process and weight updates should be performed, so that the SNN model can continually learn new knowledge while retaining old yet important information in dynamic environments.*

5. **Investigating the fault mitigation techniques for SNN systems:** Recent works studied different types of faults in SNNs and the impact of random faults on accuracy, without considering detailed fault models and the underlying SNN hardware architecture. Therefore, more studies are required to better understand the impact of hardware-induced faults in SNN systems. Then, based on these studies, cost-effective fault-mitigation techniques should be devised.

   *Objective: First, we aim to analyze the resilience of SNNs under hardware-induced faults, with a focus on approximation-induced errors, permanent faults, and transient faults. Then, we aim to investigate fault-aware training techniques that make the trained SNNs adaptive to the presence of faults. Since the fault-aware training is not applicable for all conditions (e.g., when the dataset is not fully unavailable), we hope to also develop alternate techniques (e.g., fault-aware mapping and lightweight hardware supports) to ensure that the trained SNNs are properly mapped in the SNN hardware accelerators/chips with faults, thereby maintaining the accuracy. In this manner, the yield of SNN chips can be improved with low overheads.*

## 1.5 Thesis Contributions

*This thesis aims at achieving high energy efficiency and high fault tolerance in NN-based systems, thereby enabling the deployment of advanced spiking and deep neural networks for diverse resource- and energy-constrained embedded applications.* This thesis presents novel techniques at both hardware and software levels (i.e., HW/SW-level techniques), thereby maximizing the potential for improvements offered from both domains. This thesis consists of the following contributions, as illustrated in Figure 1.7.

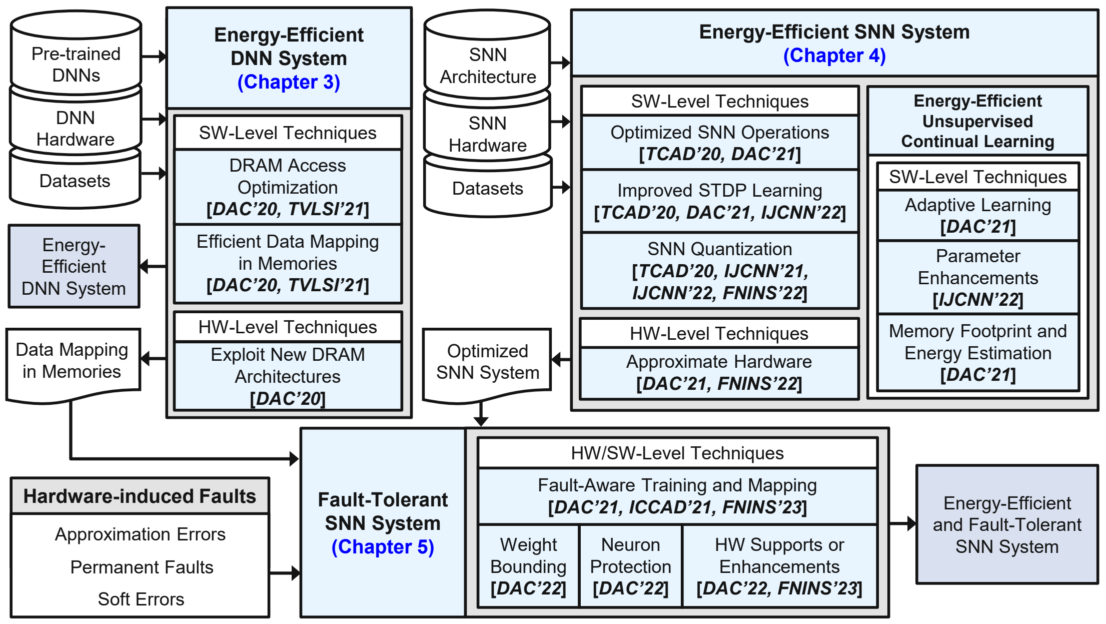

Pre-trained DNNs
DNN Hardware
Datasets
Energy-Efficient DNN System (Chapter 3)
SW-Level Techniques
DRAM Access Optimization [DAC'20, TVLSI'21]
Efficient Data Mapping in Memories [DAC'20, TVLSI'21]
HW-Level Techniques
Exploit New DRAM Architectures [DAC'20]
Energy-Efficient DNN System
Data Mapping in Memories
SNN Architecture
SNN Hardware
Datasets
Optimized SNN System
Energy-Efficient SNN System (Chapter 4)
SW-Level Techniques
Optimized SNN Operations [TCAD'20, DAC'21]
Improved STDP Learning [TCAD'20, DAC'21, IJCNN'22]
SNN Quantization [TCAD'20, IJCNN'21, IJCNN'22, FNINS'22]
HW-Level Techniques
Approximate Hardware [DAC'21, FNINS'22]
Energy-Efficient Unsupervised Continual Learning
SW-Level Techniques
Adaptive Learning [DAC'21]
Parameter Enhancements [IJCNN'22]
Memory Footprint and Energy Estimation [DAC'21]
Hardware-induced Faults
Approximation Errors
Permanent Faults
Soft Errors
Fault-Tolerant SNN System (Chapter 5)
HW/SW-Level Techniques
Fault-Aware Training and Mapping [DAC'21, ICCAD'21, FNINS'23]
Weight Bounding [DAC'22]
Neuron Protection [DAC'22]
HW Supports or Enhancements [DAC'22, FNINS'23]
Energy-Efficient and Fault-Tolerant SNN System

Figure 1.7: An overview of this PhD thesis, i.e., the integrated methodology for energy efficiency and fault tolerance for spiking and deep neural networks. The proposed techniques are highlighted in blue boxes, and the related publications are written in a ***bold-italic*** format inside a square bracket.

1. **HW/SW-level DRAM Optimization for Energy-Efficient DNN Systems**
   It proposes novel HW-level and SW-level optimization techniques for minimizing the DRAM access energy, and thereby the DNN systems' total energy. For SW-level techniques, it performs DRAM access optimization through a design space exploration (DSE) to find data partitioning and scheduling that provide the minimum number of DRAM accesses, while avoiding redundant accesses for the overlapping data during convolutional operations. It also employs efficient data mapping policies in memories to minimize the energy-per-access to off-chip DRAM and on-chip SRAM buffers. For HW-level techniques, it exploits new DRAM architectures to further optimize the DRAM energy-per-access through a generalized DRAM data mapping policy, which is proven through a DSE that considers different DRAM data mapping policies as well as different data partitioning and scheduling schemes.

2. **HW/SW-level Design and Optimization for Energy-Efficient SNN Systems**
   It proposes novel HW-level and SW-level design and optimization techniques for minimizing the memory and energy requirements of SNN systems. For SW-level techniques, it optimizes SNN operations through the elimination of an inhibitory layer and simplification of weight update operations. It also employs weight quantization to reduce the memory footprint. To make the SNN systems capable of adapting to different operational environments (i.e., non-dynamic or dynamic), it leverages the spiking activity information to determine an adaptive learning algorithm and parameter enhancements (e.g., weight decay). It also performs memory and energy estimation in DSE to quickly find an SNN model that meets the memory and energy budgets. For HW-level techniques, it employs reduced-voltage DRAMs (i.e., approximate DRAMs) to substantially reduce the operational power/energy of SNN accelerators, while minimizing the negative impact of approximation errors through fault-aware training.

3. **Cost-Effective HW/SW-level Fault Tolerance for SNN Systems**
   It proposes novel HW-level and SW-level optimization techniques for mitigating HW-induced faults, such as approximation errors, permanent faults, and transient faults (soft errors). To mitigate approximation errors in the off-chip and on-chip memories, it employs fault-aware mapping (FAM) if the training set is not fully available (e.g., due to IP or privacy reasons), and it may employ fault-aware training-and-mapping (FATM) if the training set is fully available to improve the SNN resilience. To mitigate permanent faults in the compute engine of SNN accelerators, it performs FAM-based techniques to safely map/store weights in the faulty synapses and selectively employ faulty neurons without retraining. To support this, lightweight HW enhancements are employed to perform data transformation due to the mapping technique. Meanwhile, to mitigate soft errors in the compute engine of SNN accelerators, it performs weight bounding and neuron protection using lightweight HW enhancements to ensure that the weight values and neuron behavior do not cause significant accuracy degradation.

## 1.6 Thesis Outline

The thesis is organized into six chapters. Chapter 1 introduces the need for energy-efficient and fault-tolerance spiking and deep neural networks, a summary of state-of-the-art techniques, associated research challenges, and a summary of thesis contributions. Chapter 2 presents the necessary background knowledge and work related to DNNs, SNNs, and reliability threats in NN-based computing systems. Afterward, Chapters 3, 4 and 5 discuss the concepts, techniques, and evaluations of the novel contributions of this thesis. Toward the end, Chapter 6 provides a summary of the novel contributions and findings of this thesis, as well as highlights several potential research directions for the future. A brief outline of each chapter is provided in the following.

**Chapter 2 - Background and Related Work:** This chapter describes the essential background and related work used in this thesis. It introduces the concepts of NNs in Section 2.1. Then, it explains the DNN fundamentals in Section 2.2 such as layers and operations, training and inference, network models, HW accelerators, and existing techniques for improving the energy efficiency of DNNs. Meanwhile, SNN fundamentals are explained in Section 2.3, covering neuron models, neural coding techniques, learning approaches, network architectures, training and inference, HW accelerators, and existing techniques for improving the energy efficiency of SNNs. Then, Section 2.4 describes the reliability threats in NN-based computing systems. Additionally, Section 2.5 describes quantization techniques in NNs. Meanwhile, Section 2.6 explains the DRAM fundamentals, including the organization, operations, types, and novel DRAM architectures. This chapter is concluded with a summary of the background and related work in Section 2.7.

**Chapter 3 - DRAM Optimization for Energy-Efficient DNN Systems:** This chapter discusses our novel methodology to optimize DRAM energy for enabling energy-efficient DNN systems. First, it identifies the targeted problems in Section 3.1. Then, it discusses an off-chip memory access management for DNN accelerators in Section 3.2, mainly covering techniques to reduce the redundant DRAM accesses, while exploring data partitioning and scheduling that offer the minimum number of DRAM accesses. Afterward, Section 3.3 discusses how to leverage DRAM access characteristics to devise a generalized DRAM data mapping policy that optimizes the DRAM energy-per-access and latency-per-access considering any given DRAM architectures. This chapter is concluded with a summary of the DRAM optimization for DNN systems in Section 3.4.

**Chapter 4 - Energy-Efficient SNN Systems:** This chapter discusses our novel methodology to achieve energy-efficient SNN systems. First, it identifies the targeted problems in Section 4.1. Then, Section 4.2 discusses an optimization framework for reducing the overall memory and energy requirements of SNNs. Section 4.3 discusses a systematic way to explore the quantization techniques for reducing the memory footprint of SNNs. Then, it explains how to exploit approximate hardware (e.g., DRAM) to substantially reduce the operational power/energy of SNN HW accelerators in Section 4.4. It also discusses how to enable SNNs with unsupervised continual learning capabilities considering high-precision weights (Section 4.5) and low-precision weights (Section 4.6). This chapter is concluded with a summary of energy-efficient SNN systems in Section 4.7.

**Chapter 5 - Fault-Tolerant SNN Systems:** This chapter discusses our novel methodology that employs cost-effective techniques for enabling fault-tolerant SNN systems. First, it identifies the targeted problems in Section 5.1. Then, Section 5.2 discusses how our FAM- and FATM-based techniques mitigate faults in the off-chip and on-chip memories of SNN HW accelerators. Afterward, Section 5.3 discusses how our low-cost techniques mitigate permanent faults in the compute engine of SNN HW accelerators without retraining. Meanwhile, Section 5.4 discusses how our low-cost techniques mitigate soft errors in the compute engine of SNN HW accelerators without retraining. This chapter is concluded with a summary of the fault-tolerant SNN systems in Section 5.5.

**Chapter 6 - Conclusion and Outlook:** This chapter concludes the thesis in Section 6.1 and provides an outlook for the potential future directions toward energy-efficient and fault-tolerant spiking and deep neural networks in Section 6.2.

CHAPTER 2

# Background and Related Work

This chapter presents the background knowledge and work related to NNs encompassing DNNs and SNNs, reliability threats in computing systems, quantization in NNs, and DRAM fundamentals in subsequent sections. Section 2.1 highlights a general background of the NN computation models for DNNs and SNNs. Details of DNNs are discussed in Section 2.2, including the description of DNN operations, models, training and inference process, HW accelerators, and techniques for improving the energy efficiency of DNN systems. Meanwhile, details of SNNs are discussed in Section 2.3, including the components of an SNN model (i.e., neuron models, neural coding, learning techniques, and network architectures), training and inference process, neuromorphic HW processors, and techniques for improving the energy efficiency of SNN systems. Section 2.4 presents the reliability threats in NN-based computing systems (e.g., approximation errors, permanent faults, and soft errors), and techniques for improving the fault tolerance of SNN systems against these threats. Section 2.5 presents quantization approaches for NNs. Afterward, Section 2.6 presents DRAM fundamentals as the main memory of modern computing systems, including DRAM organization, operations, and its novel architectures.

## 2.1 Introduction to Neural Networks

NNs are the prominent algorithms since they have achieved state-of-the-art accuracy for AI tasks such as image classification and object detection [LBH15]. NN algorithms can be categorized into two prominent computational models, i.e., DNNs and SNNs, which will be discussed in Section 2.2 and Section 2.3, respectively. The computational models of NNs mimic the structures and operations of a human brain, i.e., an interconnected network of neurons. Therefore, each neuron of NNs also mimics the basic functionality of a biological neuron from the brain. The functionality of a biological neuron includes several operations, i.e., the *dendrites* collect the input signals, the *cell body (soma)* computes the output signal, the *axon* transmits the output signal, and the *synapse* connects the output signal to the input of another neuron; as shown in Figure 2.1.

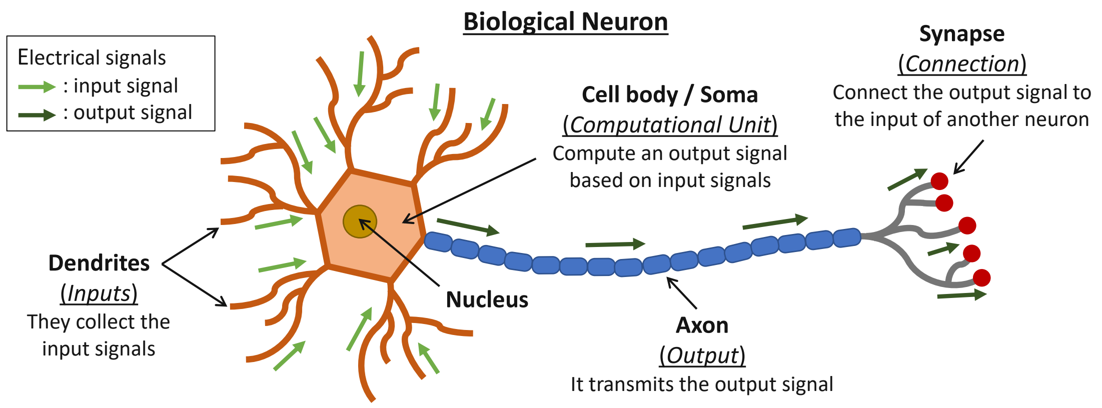

Biological Neuron
Electrical signals
: input signal
: output signal
Cell body / Soma
(Computational Unit)
Compute an output signal based on input signals
Synapse
(Connection)
Connect the output signal to the input of another neuron
Dendrites
(Inputs)
They collect the input signals
Nucleus
Axon
(Output)
It transmits the output signal

Figure 2.1: The structure of a biological neuron and its basic operations that inspire NN algorithms, encompassing DNNs and SNNs.

## 2.2 Deep Neural Networks (DNNs)

An NN is established by interconnected neurons, and these neurons are arranged in the form of layers, including an input layer, hidden layer(s), and an output layer, as shown in Figure 2.2(a). The layer arrangement aims at enabling the network to learn the hierarchical representation of input samples [LBH15]. NNs that have more than two hidden layers are typically called DNNs [B$^+$09, DY14]. The basic functionality of a neuron in DNNs can be stated as Equation 2.1 and illustrated in Figure 2.2(b).

$$O = f\left(\sum_{i=1}^{n}(w_i \cdot a_i) + b\right) \tag{2.1}$$

In Equation 2.1, $w_i$ denotes the $i^{th}$ synaptic weight, $a_i$ denotes the $i^{th}$ activation, $b$ denotes the bias, and $n$ denotes the number of input activations/weights. Meanwhile, $f(.)$ denotes the non-linear activation function to introduce non-linearity in the network, such as *Rectified Linear Unit* (ReLU).

In the basic form of DNNs, neurons are arranged in multiple layers, and each neuron in layer $l$ is connected to all neurons in layer $(l+1)$, as shown in Figure 2.2(a). This architecture is known as a *fully-connected* network. The connection between two neurons is represented by synaptic weight, which can be represented by $W^l_{x,y}$ notation. Here, $W$ denotes the synaptic weight that connects neuron $x$ in layer $(l-1)$ with the neuron $y$ in layer $l$. In this manner, each neuron and synaptic weight in a DNN can be identified.

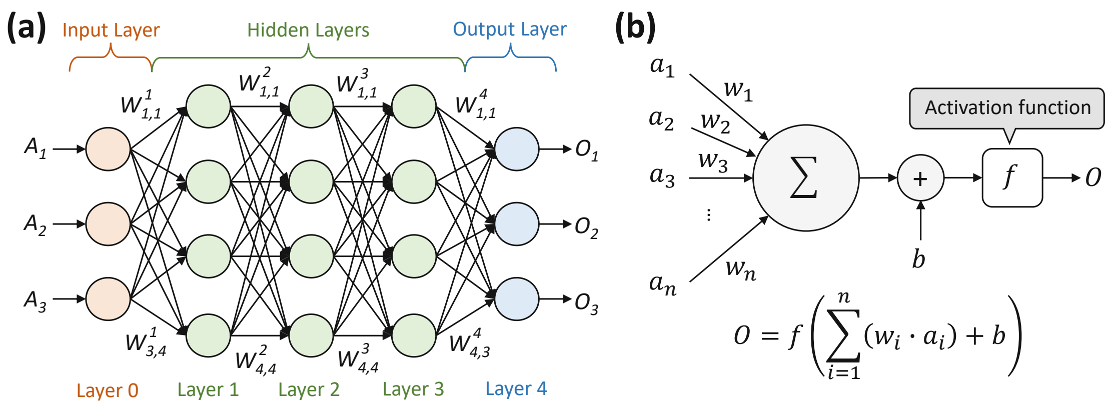


Figure 2.2: (a) An illustration of a DNN with fully-connected network architecture. (b) The functionality of a neuron in DNNs.

### 2.2.1 DNN Layers and Operations

In DNN, there are several types of layers with their specific operations. Following is a list of the most common types of DNN layers and operations.

**Fully-Connected (FC) Layer:** In this layer, each neuron in layer $l$ receives inputs from the output of all neurons in layer $(l-1)$ and their corresponding parameters, as shown in Figure 2.3(a). Therefore, the number of parameters of an FC layer is potentially huge and may require a large memory footprint. This layer is typically used for classification, such as the output layer of DNNs.

**Convolutional Layer:** In this layer, activations and weight filters are arranged in a 3D shape, and each arrangement of 2D-shaped activations is also termed as a *feature map*. Here, each neuron in layer $l$ processes selected input feature maps and weights within *receptive fields* [LBBH98b], as shown in Figure 2.3(b). This process employs multiply-and-accumulate (MAC) operations to produce output feature maps. Note, a receptive field of the convolutional layer is a portion of an input feature map that is considered by a neuron for generating an output feature (e.g., a pixel of an image). Therefore, the size of a receptive field corresponds to the 2D size of a weight filter, which is commonly termed as *kernel size*. Distance between adjacent receptive fields is called *stride*. The idea of convolution is to take advantage of the local spatial coherence in input feature maps, as processing adjacent features (e.g., pixels of an image) with smaller kernels may obtain more meaningful information than processing a whole image at once.

**Pooling Layer:** In this layer, the dimension of input feature maps is decreased through down-sampling. To do this, several adjacent features (e.g., 2x2 pixels) are grouped as a receptive field of the pooling layer, and their values are processed to produce a single output feature (e.g., 1 pixel) through a statistical approach, e.g., average value or maximum value. A prominent approach for pooling is *MaxPooling*, i.e., selecting the maximum value, as shown in Figure 2.3(c). Meanwhile, the stride size of pooling usually equals the size of the receptive field. This layer reduces the size of feature maps.

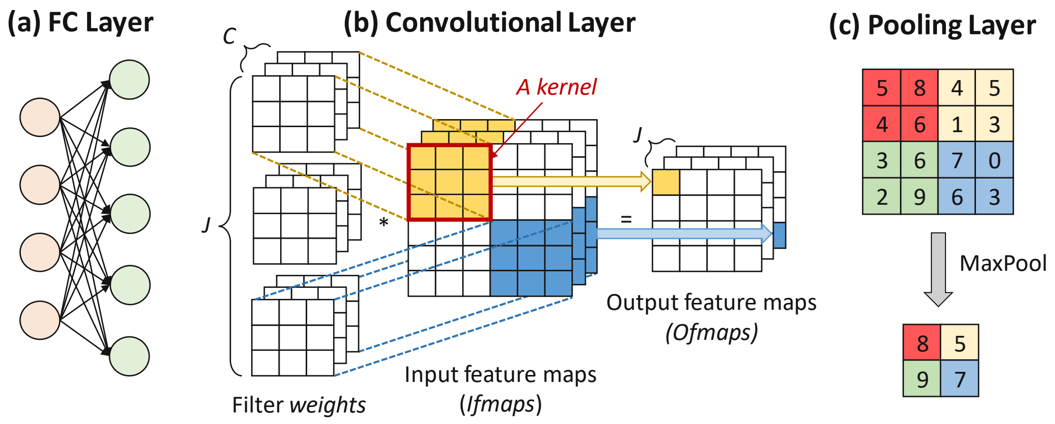


Figure 2.3: Illustrations of (a) FC layer, (b) Convolutional layer, and (c) Pooling layer.

**Normalization Layer:** This layer aims to keep the input activations of a layer having a normal distribution with the same range of values, as it eases the computations and avoids the saturation of the results from non-linear activation functions. Moreover, this layer can expedite the training process since DNN layers do not need to adapt to different distributions at each training step. The prominent normalization technique is the *Batch Normalization* [IS15].

**Activation Function:** The activation function aims to introduce a non-linearity to the network. There are several activation functions, such as *Sigmoid*, *Hyperbolic Tangent*, *Rectified Linear Unit* (ReLU), *Leaky ReLU*, *Exponential Linear Unit* (ELU), and *Softmax*, as shown in Figure 2.4. Sigmoid and Hyperbolic Tangent functions have relatively high computational complexity, hence they are rarely used in the current trends of DNN models [LBH15]. In contrast, ReLU is widely used in many DNN models, since it incurs low computational complexity yet it has an effective non-linearity impact on the networks [NH10, LBH15]. Recently, variants of ReLU such as Leaky RELU [MHN13, RDGF16] and ELU [CUH15] were developed for introducing more non-linearity to the networks, at the cost of higher computational complexity than ReLU. Meanwhile, Softmax is typically employed in the output layer of DNNs for classification purposes, as its values are within the range of (0,1) which is ideal for representing the probabilities of the output classes. Following are the mathematical definitions of these activation functions.

- Sigmoid: $y = 1/(1 - e^x)$
- Hyperbolic Tangent: $y = (e^x - e^{-x})/(e^x + e^{-x})$
- ReLU: $y = max\ (0, x)$
- Leaky ReLU: $y = max\ (\alpha x, x)$ with $\alpha$ is a small constant
- ELU: $y = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha(e^x - 1), & \text{if } x < 0 \end{cases}$ with $\alpha$ is a small constant

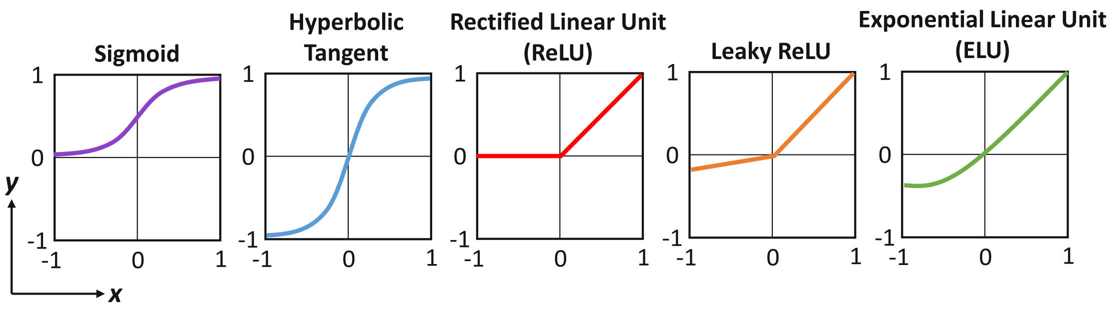


Figure 2.4: Illustrations of different activation functions: Sigmoid, Hyperbolic Tangent, ReLU, Leaky ReLU, and ELU.

### 2.2.2 DNN Training and Inference

To make a DNN model work properly to complete its task while achieving high accuracy, it has to learn the required information/knowledge. This learning period is referred to as the *training phase*. Once the training phase is complete, the learned knowledge is reflected by the DNN parameters (i.e., weights and biases). Therefore, these parameters are employed in DNN operations for completing the given task (e.g., classification, recognition, etc.) during the *inference phase*. For training and inference purposes, a dataset is employed. A dataset is typically split into three sets: *training*, *validation*, and *testing*.

- **Training set** is employed for fitting parameters (weights and biases) so that the DNN model learns patterns from training samples that can generalize well to new or unknown data. In best practices, the number of training samples is 60%-80% from all samples in the dataset.
- **Validation set** is employed for fine-tuning the hyperparameters so that the DNN model can achieve better accuracy. In best practices, the number of validation samples is 10%-20% from all samples in the dataset.
- **Testing set** is employed for testing the accuracy of DNN inference with unbiased evaluation. In best practices, the number of testing samples is 10%-20% from all samples in the dataset.

There are several approaches for training DNNs, i.e., *supervised learning*, *unsupervised learning*, and *reinforcement learning*. These approaches are described in the following.

- **Supervised Learning:** It is the most common approach for training DNNs, as it achieves state-of-the-art accuracy for various ML tasks. It employs *labeled data samples* to train DNNs, and the prominent algorithm for this learning approach is *gradient backpropagation* (or simply *backpropagation*) [SCYE17]. Here, the training set is typically divided into *batches*, and each batch of samples is passed through the network for updating the parameters. The outputs of the network are compared with the labels to evaluate the distance between the outputs with the expected values (i.e., *loss score*) using a *loss function*. The loss score is used to compute the *gradients* of all network parameters (weights and biases). These gradients are then used to adjust the corresponding weights and biases so that the loss score is minimized. Note, a complete pass of the entire training set is referred to as an *epoch*, and the number of epochs for proper training depends on the complexity of the task.

- **Unsupervised Learning:** This learning approach employs *unlabeled data samples* for training the networks. Here, the prominent technique is *clustering* whose idea is to find common similarities/patterns in training samples [SCYE17]. Common DNNs that employ unsupervised learning are Generative Adversarial Networks (GANs) and Autoencoders. Furthermore, unsupervised learning is also beneficial for systems that require online training with unlabeled data samples (taken directly from environments) to update their knowledge at run time, such as exploratory mobile agents/robots.
- **Reinforcement Learning:** This learning approach employs *unlabeled data samples* and *rewards* to effectively train the networks. Its idea is to employ agent that makes decision on what action to take [SCYE17]. If the action leads to the expected result, then the agent will get a reward. Otherwise, the agent will not get any reward. In this manner, a series of actions that leads to the highest reward is the final knowledge.

During the training phase, there are two prominent issues that may occur, i.e., *underfitting* and *overfitting*. Underfitting refers to the problem when the network model does not effectively learn the important features from the training samples. Meanwhile, overfitting refers to the problem when the network model learns both the important features and the noise, hence the corresponding model cannot generalize well for new or unseen data samples. It is usually indicated by a condition when the training error is low and test error is high [B+09]. The most prominent technique for addressing overfitting is the *early stopping strategy*. Its key idea is to quickly detect overfitting, and if so, it stops the training process. Overfitting detection is usually performed by employing the validation set to evaluate how well the model can generalize after training.

### 2.2.3 DNN Models

Current trends show that, researchers have been continuously developing DNN models to achieve higher accuracy. One of the main indicators is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which evaluates the accuracy of DNN models for the ImageNet dataset [DDS+09]. Convolutional Neural Networks (CNNs), a particular type of DNNs, have emerged as a prominent design approach for developing deep learning models [PHS21b], since this design approach has achieved state-of-the-art accuracy. Following are some prominent CNN models from the community.

- **LeNet [LBBH98b]:** It is one of the earliest CNN models that were trained with the backpropagation technique. It consists of 2 convolutional layers and 3 FC layers.
- **AlexNet [KSH12]**: This model won the ILSVRC 2012 and outperformed state-of-the-art computer vision techniques. Therefore, AlexNet is often considered the pioneer

of high-accuracy CNNs that started a deep learning era. This model consists of a sequence of 5 convolutional layers and 3 FC layers. It also introduced the ReLU activation function to add non-linearity to the model.

- **GoogLeNet [SLJ+15]:** This model was developed by Google and won the ILSVRC 2014. It has 22 layers including convolutional layers, FC layers, pooling layers, and inception modules. The inception module allows the model to use multiple filter sizes in the same layer, whose outputs are concatenated and passed to the next layer. Hence, the inception module can extract input features at varying scales in the same layer.

- **VGG-16 [SZ14]:** This model was the runner-up of the ILSVRC 2014. VGG-16 became one of the most popular image classification models due to its idea of employing many hidden layers to improve accuracy. It consists of 13 convolutional layers, 3 FC layers, and 5 pooling layers. It also employs the ReLU activation functions and the MaxPooling layers. Afterward, it was realized that adding more hidden layers may cause the model to suffer from vanishing or exploding gradients.

- **ResNets [HZRS16]:** Researchers from Microsoft developed the Residual Networks (ResNets) to address the problem of vanishing or exploding gradients. Its idea is to reformulate the layers as learning functions with reference to the layer inputs (so-called *residue*) for preserving the reference information across layers. Therefore, the number of hidden layers can be increased without facing vanishing or exploding gradients.

- **SqueezeNet [IHM+16]:** This model was developed for achieving AlexNet-level accuracy with a significantly lower number of parameters (50× smaller). It employs fire modules that are slightly different from conventional CNNs. Each fire module has a *squeeze layer* (i.e., a convolutional layer with 1x1 filters) and an *expand layer* (i.e., a convolutional layer with 1×1 and 3×3 filters).

- **MobileNet [HZC+17]:** This model was developed for mobile and embedded vision applications. It employs depthwise separable and pointwise convolutions to factorize a standard convolutional layer, thereby reducing the computations and the model size as compared to the standard convolutional layer.

### 2.2.4 DNN Hardware Accelerators

To expedite the DNN inference, DNN HW architectures typically employ massively parallel *processing elements* (PEs), that perform basic DNN operations (i.e., MAC). They can be classified as temporal and spatial architectures [SCYE17]. In *temporal architectures*, the control unit is centralized, and the PEs can only access data through the central/global memory as there are no connections among PEs. Examples of this category are CPUs and GPUs. Meanwhile, in *spatial architectures*, each PE employs local memory and control units, and there are connections among PEs as well. Examples of this category are specialized DNN HW accelerators, which are tailored and optimized for specific dataflows. There are several dataflow types explored in the literature.

- **Weight Stationary:** It aims at reducing the energy consumption of off-chip accesses for weights. To do this, it first stores and keeps the weights in the local memory of PEs. Then, it moves the input activations across PEs to generate output partial sums, which are also propagated to other PEs.

- **Output Stationary:** It aims at reducing the energy consumption of off-chip accesses for partial sums. To do this, it moves the input activations and weights across PEs to generate partial sums, which are then kept in the local PE memory.

- **Row Stationary:** It aims at reducing the energy consumption of off-chip accesses for all data types (i.e., weights and partial sums). To do this, it assigns the processing of a 1D row convolution into each PE. Here, it keeps the row of filter weights stationary inside the local memory, then it streams the row of input activations into the PEs for generating partial sums.

- **No Local Reuse:** It aims at maximizing the storage capacity while minimizing the off-chip memory bandwidth. To do this, no local storage is provided, hence there is no data reuse at the PE level. Instead, it moves all data types (i.e., weights and partial sums) across PEs to complete the processing.

In recent years, larger and deeper DNNs have been developed to achieve higher accuracy. However, large DNN models require a huge memory footprint, intensive computations, and energy consumption. Therefore, *to maximize the performance (i.e., speed) and energy efficiency of DNN inference, specialized DNN HW accelerators are employed* [SCYE17, CBM+20c]. In the literature, there is a significant amount of work on DNN accelerator designs. Some of these accelerators aim at expediting the *un-structurally sparse networks* by exploiting sparse activations and/or weights. Activation sparsity usually comes from the employment of ReLU activation function, while weight sparsity comes from the employment of pruning. Popular examples of these DNN accelerators are Cambricon-X [ZDZ+16], Cnvlutin [AJH+16], EIE [HLM+16], ZeNa [KAY17], SCNN [PRM+17], Bit-Pragmatic [ADJ+17], UCNN [HYA+18], Bit Fusion [SPS+18], SparTen [GCTV19], SqueezeFlow [LJG+19], Laconic [SLM+19a]. However, recent studies show that *simply employing sparsity does not directly lead to energy efficiency improvements for DNN inference*, as the sparsity requires more sophisticated and complex accelerator designs to achieve high performance, thereby requiring larger area and higher power/energy consumption [CYES18a, YLP+17]. Meanwhile, the other accelerator designs aim at expediting *dense networks*, such as DianNao [CDS+14], ShiDianNao [DFC+15], Eyeriss [CES16], DaDianNao [LLL+16], Tensor Processing Unit (TPU) [JYP+17], FlexFlow [LYL+17], DNA [TYO+17], MAERI [KSK18], and DNPU [SLL+18]. Furthermore, these accelerators can also be used for expediting the *structurally sparse networks* by tailoring the processing dataflows to the respective accelerator architectures [AHS17, YLP+17]. All these DNN accelerators have similar characteristics: (1) massively parallel compute arrays to perform MAC operations, and (2) dedicated memory hierarchy to maximize local data reuse while minimizing costly off-chip memory accesses.

Among the existing DNN accelerators, TPU [JYP+17] is one of the most prominent designs due to its high performance and efficiency, hence its design has inspired many other DNN accelerators [HPT+18]. TPU was developed by Google to expedite the DNN processing for cloud-based applications. The core design of TPU is the systolic array (SA)-based compute engine (i.e., "SA engine" for brevity) that employs 256×256 processing elements (PEs). Each PE in the SA engine performs three main tasks, i.e., receiving data from the upstream neighbor, performing MAC operation, and passing the data along with the partial sum to the downstream neighbor. To support this process, the SA engine of TPU employs a *weight stationary* dataflow. *In this thesis, most of the case studies for DNN accelerators are based on a similar design*, as shown in Figure 2.5. The overview of DNN HW accelerator architecture is shown in Figure 2.5(a) and the architecture of the SA engine is illustrated in Figure 2.5(b). Meanwhile, the architecture of each PE is presented in Figure 2.5(c).

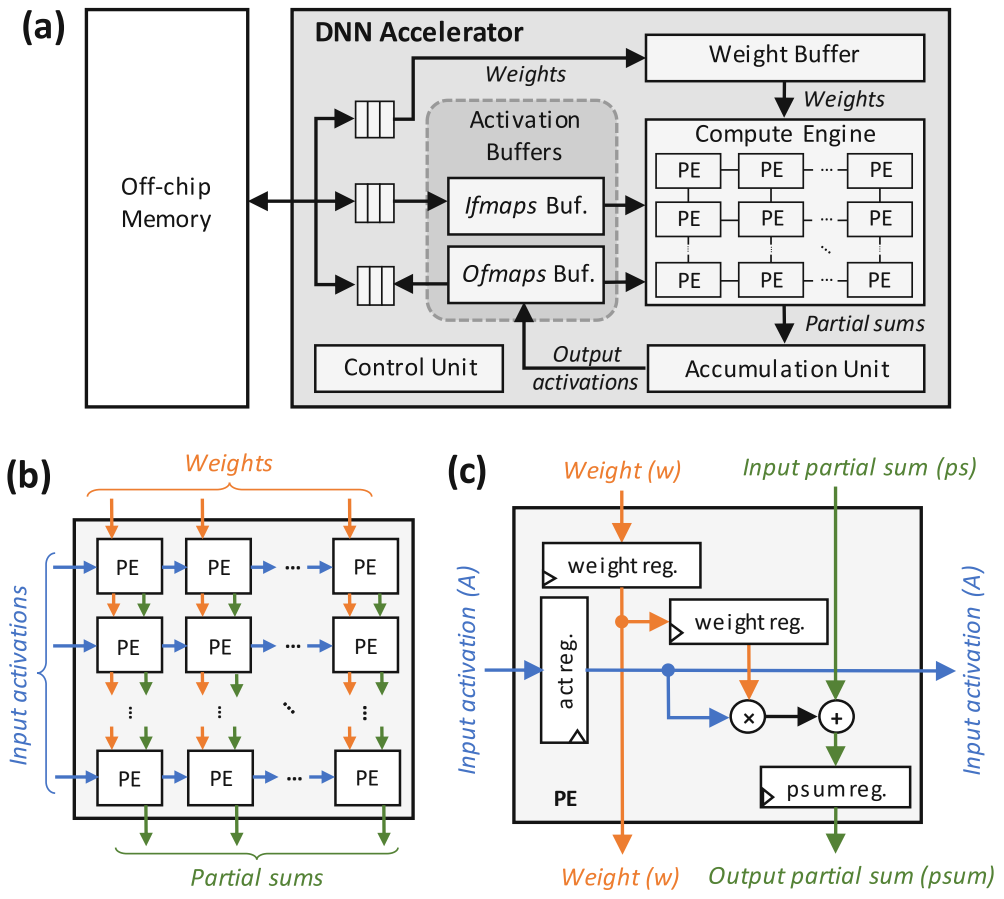


Figure 2.5: (a) An overview of the DNN HW accelerator. Here, the activation buffers consist of a buffer for input feature maps (*ifmaps buffer*) and a buffer for output feature maps (*ofmaps buffer*). (b) A detailed view of the systolic array-based compute engine architecture. (c) A detailed view of the processing element architecture.

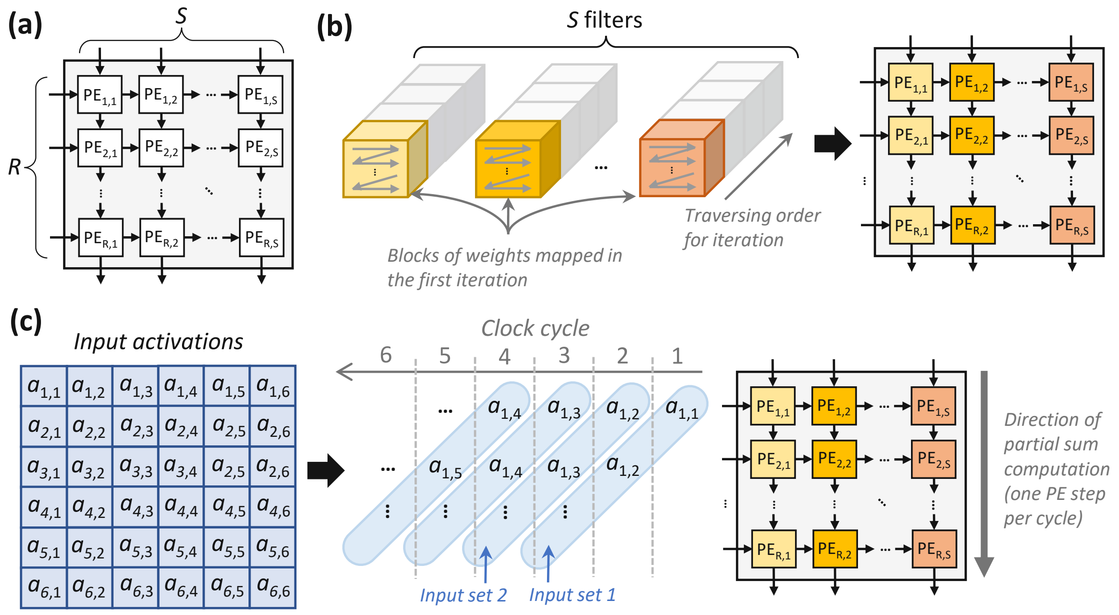

(a)
S
R
PE1,1
PE1,2
PE1,S
PE2,1
PE2,2
PE2,S
PER,1
PER,2
PER,S
(b)
S filters
Blocks of weights mapped in the first iteration
Traversing order for iteration
(c)
Input activations
Clock cycle
6 5 4 3 2 1
Input set 2
Input set 1
Direction of partial sum computation (one PE step per cycle)

Figure 2.6: Overview of the dataflow for mapping data to the SA-based engine and performing convolutional computations.

Figure 2.6 presents the overview of the dataflow for mapping data to the SA engine and performing convolutional computations. If we consider an array of PEs with $R \times S$ size (with $R$ and $S$ denoting the number of rows and columns of PEs, respectively), then only a set of $R \times S$ weights from the weight buffer can be mapped on the SA engine at one time. The dataflow maps a specific filter to a column of the SA engine at one time, hence only $R$ number of weights from the same filter are mapped. Once the weights are mapped to the array, they are kept stationary. Then, the input activation values are fed from the left side of the SA engine. In each PE, the input activation is computed with the stored weight through MAC operation to generate the partial sum. To ensure that PEs can operate in lockstep to generate the correct partial sums, the input activations must be arranged properly, as shown in Figure 2.6.

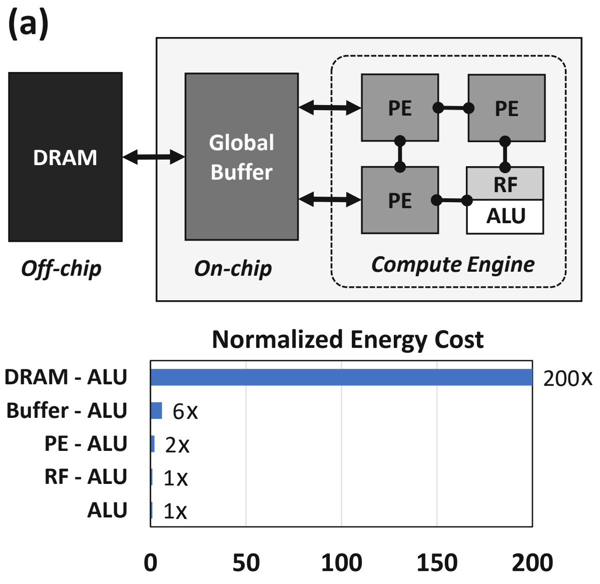
In the first clock cycle, the $\mathrm{PE}_{1,1}$ performs multiplication between the first input activation from the first input set and the first weight from the first filter, then stores the generated partial sum in the partial sum register. In the second clock cycle, $\mathrm{PE}_{2,1}$ receives the second activation from the first input set, multiplies it with the second weight from the first filter, adds the multiplication product with the partial sum from $\mathrm{PE}_{1,1}$, and then stores the generated partial sum in the partial sum register. Furthermore, in the same clock cycle, $\mathrm{PE}_{1,1}$ receives the first activation from the second input set and multiplies it with the stored weight (i.e., the first weight from the first filter); while $\mathrm{PE}_{1,2}$ receives the first activation from the first input set, and multiplies it with the stored weight (i.e., the first weight from the second filter). In this manner, the SA generates its first partial sum from the first column after $R$ clock cylces, and can generate up to $S$ partial sums simultaneously. If the number of weights in a filter is more than the number of rows in the array ($R$), then the filter needs to be divided into blocks, and each block has a maximum of $R$ weights. Furthermore, the partial sums from these blocks must be accumulated using the accumulation units to generate the final output activation. Meanwhile, if the number of filters is more than the number of columns in the array ($S$), then the filters need to be divided into sets, and each set has a maximum of $S$ filters.

**DNN Accelerators with Conventional and Emerging Technologies:** In conventional computing paradigm, DNN accelerators employs the von-Neumann architecture with conventional CMOS technology. Since DNN processing is data intensive, this computing paradigm leads to frequent data movements between compute engines and memory modules. To minimize data movements, recent works developed DNN accelerators with non-von-Neumann architecture and emerging technologies, such as Resistive RAM (RRAM), Magnetic RAM (MRAM), and Carbon Nanotube FET (CNTFET). Here, the non-volatile memory (NVM) devices like RRAM and MRAM are employed to store weights [AMY+23]. Meanwhile, CNTFET is employed to replace CMOS for other circuit parts (e.g., logic) due to its better energy-delay-product (EDP) and scalability [CSLC21]. However, these emerging NVM technologies still face reliability issues, such as variability (resistance drift) and endurance, indicating that they are not yet mature and still need further studies [AZH+23]. Moreover, the conventional computing with off-chip DRAM and on-chip SRAM structure have been widely used in real-world applications. Therefore, *in this thesis, we focus on the conventional computing paradigm with CMOS technology for DNN accelerators to provide immediate impact in both academia and industry.*

### 2.2.5 Techniques for Improving the Energy Efficiency of DNNs

To expedite the inference process, many DNN HW accelerators have been designed and employed in the past few years since they can provide higher performance efficiency as compared to the general-purpose CPU and GPU-based solutions. However, the energy consumption of these accelerators is typically dominated by the off-chip memory accesses, especially when the full DNN parameters and processing cannot be mapped at the same time to the accelerator fabric. The reason is that, the size of DNN models is large, while the typical size of compute engines and on-chip memory are small [SCYE17]. Therefore, the on-chip resources of a DNN accelerator are usually not sufficient to hold and process a complete DNN model or even one layer of the network at one time. Moreover, each data value is usually involved in multiple computations (i.e., MAC operations). Therefore, multiple redundant accesses for the same data to the off-chip memory (i.e., DRAM) are required to complete the DNN processing. However, such redundant accesses hinder the DNN accelerators from gaining further energy efficiency improvement as DRAM access energy is significantly higher as compared to other operations in DNN accelerators [SCYE17, PHS20, CBM+20b, AAH+20, PHS21b], which is also shown in Figure 2.7. In summary, *reduction of DRAM access energy is the key for improving the energy efficiency of DNN-based systems.*

Many techniques have been proposed to optimize the DRAM access energy, as it is the



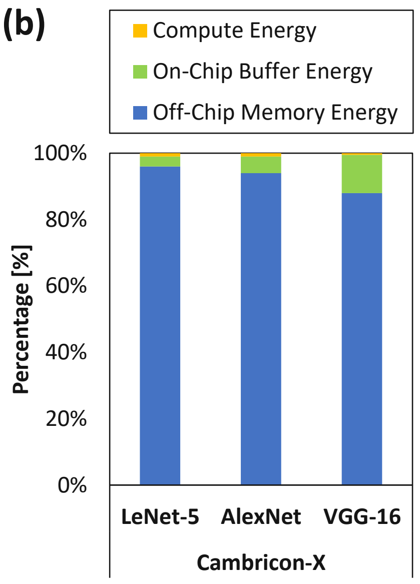


Figure 2.7: (a) Energy consumption of data movement considering the memory hierarchy of a DNN HW accelerator (adapted from [SCYE17]), showing that DRAM energy-per-access is significantly higher than other operations. Note, DRAM fundamentals (e.g., organization and operations) are discussed in Section 2.6. (b) Breakdown of energy consumption of Cambricon-X [ZDZ+16], showing that DRAM access energy dominates the energy consumption of DNN accelerators.

key to improving the energy efficiency of DNN systems, as summarized in Table 2.1. Some works compress the DNN model size with an expectation of a reduced number of DRAM accesses, and thereby the systems' total energy [DLH+20] through pruning (i.e., unstructured [HMD16, LKD+16, MHMS18] and structured [AHS17, HLL+18]), and quantization [GAGN15, HMD16, JKC+18a] whose concept is discussed in Section 2.5. However, recent studies show that only relying on *model compression* does not directly lead to energy savings, as it may incur high overhead due to additional operations and resources (e.g., data encoding and decoding) [YLP+17, CYES18b, KMZ19]. Other works employ *a data partitioning and scheduling* approach to minimally move the data from DRAM to on-chip memory and then reuse the data multiple times for computation, i.e., fixed scheduling [ZLS+15, ZSF+19] and adaptive scheduling [LYL+18, TKP20]. This approach can be combined with quantization and structured pruning to further improve energy efficiency [PHS21b]. However, there are several limitations to the existing works as discussed in the following.

Table 2.1: An overview of state-of-the-art works for improving the energy efficiency of DRAM accesses in DNN systems [PHS21b].

| **Approach** | **Technique** | **Related Work** | **Brief Description** | **Benefits** | **Cost / Weaknesses** |
|---|---|---|---|---|---|
| Model Compression | Unstructured Pruning | [HMD16] [LKD+16] [MHMS18] | It removes redundant parameters in a DNN based on their significance, thereby reducing the DNN model size (i.e., memory footprint) and the number of DNN computations. | • Less memory footprint<br>• Less number of computations | • Accuracy loss<br>• Training cost<br>• Data encoding |
| | Structured Pruning | [AHS17] [HLL+18] | It removes redundant groups of parameters in a DNN based on their significance, while considering the location of the parameter groups of to be removed. | • Less memory footprint<br>• Less number of computations<br>• Structured network | • Accuracy loss<br>• Training cost<br>• Data encoding |
| | Quantization | [GAGN15] [HMD16] [JKC+18a] | It reduces the precision of DNN parameters and intermediate outputs by converting them from floating-point to fixed-point format. | • Less memory footprint<br>• Simpler hardware modules | • Accuracy loss<br>• Training cost |
| Data Partitioning and Scheduling | Fixed Scheduling | [ZLS+15] [ZSF+19] | It employs static data partitioning factors and scheduling across layers of a DNN, by giving priority of reuse to only one specific data type, either input feature maps (*ifmaps*), output feature maps (*ofmaps*), or weights. | • Less memory accesses for a specific data type | • Compile time |
| | Adaptive Scheduling | [LYL+18] [TKP20] | It employs adaptive data partitioning factors and scheduling across layers of a DNN, by giving priority of reuse to a data type that has highest reuse factors at each layer. | • Less memory accesses for all data types | • Compile time |

**Limitations of the State-of-the-Art Works:**

1. *Optimization of the number of DRAM accesses:* The data partitioning and scheduling are required to determine the portion of data to be accessed from DRAM for on-chip computation. Here, there are overlapping data in convolutional operations that should not be re-fetched again from DRAM to minimize the DRAM accesses. However, this aspect is not optimized by state-of-the-art works. Therefore, their analytical models for estimating the number of DRAM accesses provide sub-optimal results and need to be reformulated.

2. *Optimization of the DRAM energy-per-access:* DRAM access energy is dependent on the number of accesses and the energy-per-access that varies depending upon whether the access faces a row buffer hit, a row buffer miss, or a row buffer conflict. Therefore, the DRAM energy-per-access should also be optimized to get further energy savings. However, this aspect is not optimized by state-of-the-art works.

3. *Effective data mapping in the on-chip memory:* To efficiently move the data between the DRAM and the compute engine, a judicious data mapping in the on-chip memory is required. Otherwise, the data accessed from DRAM might not be utilized in an energy-efficient manner by the compute engine, thereby decreasing the energy saving benefits. However, this aspect is not considered in state-of-the-art works.

## 2.3 Spiking Neural Networks (SNNs)

SNNs are considered the third generation of NN computation models because they exhibit high biological plausibility by mimicking the biological brain through the employment of spiking networks as well as spikes (i.e., *action potentials*) to convey information [Maa97]. Recently, with the advances in neuromorphic computing, SNNs have demonstrated great success in achieving high accuracy with ultra-low power/energy consumption, thereby making SNNs suitable for resource-constrained computing systems. SNNs can achieve high accuracy due to the effectiveness of their learning mechanism, and ultra-low power/energy consumption due to their sparse spike-based computations. An SNN model is composed of several design aspects, i.e., *network architecture*, *spiking neuron model*, *neural coding*, and *learning rule* [MGNDM19]. Figure 2.8 shows the overview of an SNN architecture. To perform SNN processing, each input sample (e.g., an image) is first converted into sequences of spikes (so-called *spike trains*) using a specific neural coding. Here, each data from an input sample (e.g., a pixel of an image) is mapped to a specific neuron in the input layer, and this neuron generates a spike train that represents the corresponding data value (e.g., a pixel value). Afterward, these input spikes are fed to the SNN model with a specific architecture, as shown in Figure 2.8(a). In each neuron, input spikes trigger the increase of neurons' membrane potential, and the respective neuron generates output spikes when the membrane potential reaches the neurons' membrane threshold potential. A neuron can generate spikes and pass them to another neuron (the destination neuron) through a synapse connection. These spikes are seen as input spikes (i.e., *presynaptic spikes* when they have not reached the synapse) by the destination neuron. Afterward, the destination neuron will process these spikes and may generate the output spikes (i.e., *postsynaptic spikes*), as shown in Figure 2.8(b). In the output layer, e.g., the excitatory layer in Figure 2.8(a), the output spikes are decoded based on the neural coding to understand the information.

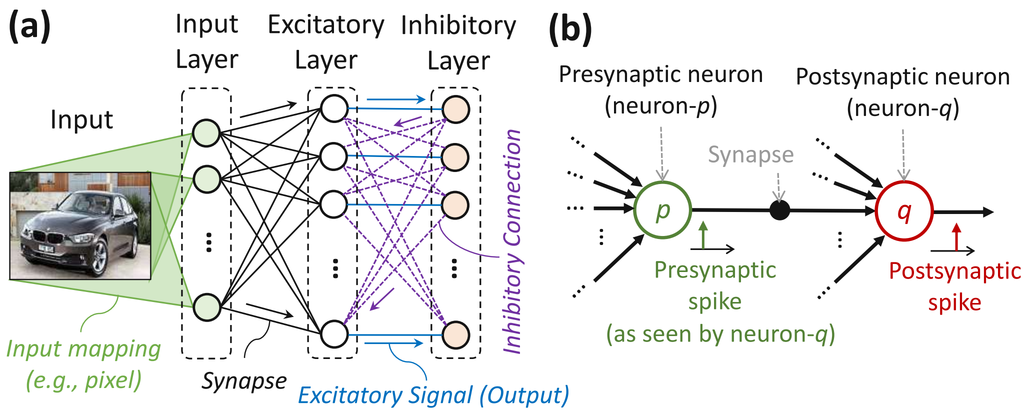

(a)
Input Layer
Excitatory Layer
Inhibitory Layer
Input
Inhibitory Connection
Input mapping (e.g., pixel)
Synapse
Excitatory Signal (Output)
(b)
Presynaptic neuron (neuron-p)
Postsynaptic neuron (neuron-q)
Synapse
p
q
Presynaptic spike (as seen by neuron-q)
Postsynaptic spike

Figure 2.8: (a) Illustration of an SNN with a fully-connected network architecture that supports biologically-plausible learning rules. (b) Illustration of a single synaptic connection between neuron-$p$ and neuron-$q$, showing different components: presynaptic and postsynaptic neurons, presynaptic and postsynaptic spikes from the perspective of neuron-$q$, and a synapse.

### 2.3.1 Spiking Neuron Models

The spiking neuron is the basic building block of SNNs since it is where most of SNN processing happens [GK02, Izh04]. The behavior of a spiking neuron is called the *neuronal dynamics* [GKNP14]. The basic neuronal dynamics of a neuron is that the neuron will increase its membrane potential ($v_{mem}$) when a spike comes. The magnitude of 'the membrane increase' corresponds to the strength of the connecting synapse, i.e., weight ($wgh$). If the $v_{mem}$ reaches/surpasses the membrane threshold potential ($v_{th}$), the neuron generates an output spike. Afterward, the $v_{mem}$ goes to the reset potential ($v_{reset}$). The $v_{reset}$ is usually equal to the resting potential ($v_{rest}$), i.e., the potential when the neuron does not perform operations for a long period of time. Note, details of the neuronal dynamics may differ depending on the *neuron model*, and there are different types of neuron models as shown in Table 2.2. Among them, the prominent spiking neurons are described in the following.

- **Hodgkin-Huxley Model** [HH52]: It is considered as the most biologically-plausible neuron model and the most complex one. As a consequence, the development of SNNs using this model is costly/inefficient. This model considers three types of ion channels, i.e., sodium ($Na$), potassium ($K$), and a leak current that mainly consists of $Cl^-$ ($L$). Its neuronal dynamics can be generally formulated as Equations 2.2-2.3, which can be extended further for achieving more accurate behavior. In these equations, $v_{mem}(t)$

Table 2.2: A representative list of neuron models (adapted from [CMA+13]).

| Neuron Model | Year | Reference |
|---|---|---|
| Integrate-and-Fire (IF) | 1907 | [Abb99] |
| McCulloch-Pitts | 1943 | [MP43] |
| Hodgkin-Huxley | 1952 | [HH52] |
| Perceptron | 1958 | [Ros58] |
| Fitzhugh-Nagumo | 1961 | [Fit61] |
| Leaky Integrate-and-Fire (LIF) | 1965 | [Ste65] |
| Morris-Lecar | 1981 | [ML81] |
| Quadratic IF | 1986 | [EK86] |
| Hindmarsh-Rose | 1989 | [RH89] |
| Spike Response Model | 1995 | [Ger95] |
| Time-varying IF | 1998 | [SZ98] |
| Wilson Polynomial | 1999 | [Wil99] |
| IF-or-Burst | 2000 | [SCSR00] |
| Resonate-and-Fire | 2001 | [Izh01] |
| Izhikevich | 2003 | [Izh03] |
| Exponential IF | 2003 | [FTHVVB03] |
| Generalized IF | 2004 | [JLG04] |
| Adaptive Exponential IF | 2005 | [BG05] |
| Mihalas-Neibur | 2009 | [MN09] |
| Augmented LIF | 2013 | [CMA+13] |

denotes the membrane potential at time $t$, $C_{mem}$ denotes the membrane capacitance, $I(t)$ denotes the input current, and $\sum_k I_k$ denotes the sum of the ionic currents which pass through the membrane. $g_{Na}$, $g_K$, and $g_L$ denote the maximum conductance for $Na$, $K$, and $L$, respectively. $m$, $n$, and $h$ denote the gating variables to model the probability of the respective channel to open at the given time. Meanwhile $E_{Na}$, $E_K$, and $E_K$ denote the reversal potential of the respective channel.

$$C_{mem}\frac{dv_{mem}(t)}{dt} = -\sum_k I_k(t) + I(t) \tag{2.2}$$

$$\sum_k I_k = g_{Na}m^3h(v_{mem}(t) - E_{Na}) + g_K n^4(v_{mem}(t) - E_K) + g_L(v_{mem}(t) - E_L) \tag{2.3}$$

- **Integrate-and-Fire (IF) Model** [Fal19]: It is considered the simplest neuron model, as it incurs the lowest computation requirements as compared to other neuron models [Izh04]. This model increases the $v_{mem}$ proportionally to the connecting weight each time a spike arrives at the neuron. If the $v_{reset}$ reaches/surpasses the $v_{th}$, the neuron generates an output spike, and then it goes back to the $v_{reset}$. Typically, the $v_{reset}$ is equal to the $v_{rest}$. Afterward, the neuron will not be able to generate spikes in a certain period, i.e., *refractory period* ($t_{ref}$), before becoming active again to perform operations. Note, in some implementations, the IF neuron model may not consider $t_{ref}$. Its neuronal dynamics can be formulated as Equations 2.4-2.5.

$$C_{mem}\frac{dv_{mem}(t)}{dt} = I(t) \tag{2.4}$$

$$\text{if} \quad v_{mem} \geq v_{th} \quad \text{then} \quad v_{mem} \leftarrow v_{reset} \tag{2.5}$$

- **Leaky Integrate-and-Fire (LIF) Model** [GK02]: It is an enhanced version of the IF neuron model by employing the concept of *leaky membrane potential* and *refractory period* $t_{ref}$ to achieve higher biological plausibility, as shown in Figure 2.9. This model increases the $v_{mem}$ proportionally to the connecting weight each time a spike arrives at the neuron. If there are no input spikes, the $v_{mem}$ leaks at a certain rate. If the $v_{mem}$ reaches the $v_{th}$, the neuron generates an output spike, and then it goes back to the $v_{reset}$. Typically, the $v_{reset}$ is equal to the $v_{rest}$. Afterward, the neuron will not increase its $v_{mem}$ within the $t_{ref}$ even if there are spikes received in the input. Once the $t_{ref}$ passes, the neuron is active again to perform its operations. Its neuronal dynamics can be modeled using a resistance-conductance (RC) circuit and formulated as Equations 2.6-2.8. Here, $R_{mem}$ denotes the membrane resistance, and $\tau_{leak} = R_{mem}C_{mem}$ denotes the time constant for the '$v_{mem}$ leak'. *In this thesis, most of the case studies for neuron model consider the LIF neuron model, as LIF can provide reasonable bio-plausible spike patterns with low computational cost.*

$$I(t) = \frac{v_{mem}(t) - v_{reset}}{R_{mem}} + C_{mem}\frac{dv_{mem}(t)}{dt} \tag{2.6}$$

$$\tau_{leak}\frac{dv_{mem}(t)}{dt} = -[v_{mem}(t) - v_{reset}] + R_{mem}I(t) \tag{2.7}$$

$$\text{if} \quad v_{mem} \geq v_{th} \quad \text{then} \quad v_{mem} \leftarrow v_{reset} \tag{2.8}$$

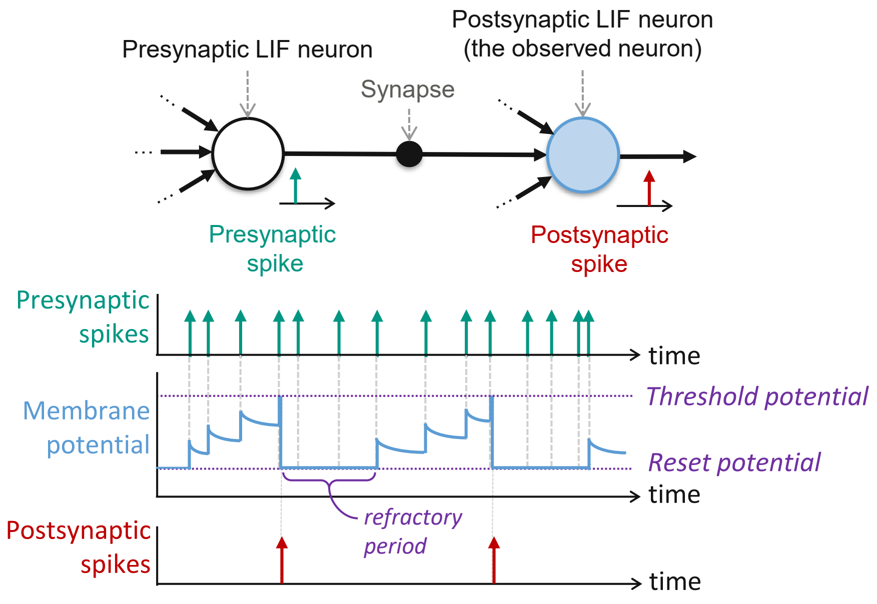


Figure 2.9: Illustration of the neuronal dynamics of the LIF neuron model.

- **Izhikevich Model** [Izh03]: It can reproduce around 20 types of spiking patterns like the Hodgkin-Huxley model, while incurring less computational intensity [Izh04]. As compared to the IF/LIF model, the Izhikevich model is more biologically plausible while incurring higher computational intensity. Its neuronal dynamics can be formulated as Equations 2.9-2.11. Here, $u_{mem}$ denotes the membrane recovery variable, $a$ denotes the time scale of the $u_{mem}$ (i.e., a smaller value leads to slower recovery), and $b$ denotes the sensitivity of the $u_{mem}$ to the fluctuations of the $v_{mem}$.

$$\frac{dv_{mem}(t)}{dt} = 0.04v_{mem}^2(t) + 5v_{mem}(t) + 140 - u_{mem}(t) + I(t) \tag{2.9}$$

$$\frac{du_{mem}(t)}{dt} = a(bv_{mem}(t) - u_{mem}(t)) \tag{2.10}$$

$$\text{if} \quad v_{mem} \geq v_{th}, \quad \text{then} \begin{cases} v_{mem} \leftarrow v_{reset} \\ u_{mem} \leftarrow u_{mem} + d \end{cases} \tag{2.11}$$

### 2.3.2 Neural Coding Techniques

Neural coding (a.k.a. *spike coding* [PS20] or *information coding* [MGNDM19]) converts each information/data (e.g., a pixel of an image) into a spike train. In the literature, multiple types of neural coding techniques have been studied [AHMK21, GFES21], and their descriptions are presented in the following.

- **Rate Coding (a.k.a. Frequency Coding)**: It utilizes the frequency of spikes (rate) to encode data. It converts the intensity of a data value into a spike train, i.e., a higher intensity is typically converted into a higher number of spikes than a lower intensity, as shown in Figure 2.10(a). Most SNN works employ the rate coding since it is simple, relatively robust against noise [DC15], and has demonstrated high accuracy when employed in SNNs under unsupervised learning settings, which is beneficial for efficient online learning capabilities of autonomous systems (e.g., mobile robots, UAVs, and UGVs) [PS21b, PS22a, PS23a, PS23b]. Rate coding has several variants: (1) *spike count*, which counts the number of spikes over time, hence having the most efficient mechanism than other variants; (2) *spike density*, which averages the number of spikes over several runs; and (3) *population activity*, which averages over several neurons that act together on the same stimulus [AES21]. *In this thesis, most of the case studies for neural coding consider the rate coding with the spike count scheme.*

- **Temporal Coding**: It utilizes temporal information to encode data and may achieve lower power consumption than the rate coding if they employ a lower spiking rate within the same/smaller time window [GT98]. There are several variants of temporal coding techniques (i.e., *burst*, *time-to-first spike*, *phase*, and *rank-order coding* schemes) as described in the following.

– **Burst Coding** [PKCY19]: It encodes data in the form of burst spikes, i.e., a sequence of spikes that has a short *inter-spike interval* (ISI). To do this, a higher-intensity data value is typically converted into a shorter ISI than lower-intensity data, as shown in Figure 2.10(b).
– **Time-To-First Spike (TTFS) Coding** [PKCY19]: It encodes data in the form of burst spikes, i.e., a sequence of spikes that has a short *inter-spike interval* (ISI). To do this, a higher-intensity data value is typically converted into a shorter ISI than lower-intensity data, as shown in Figure 2.10(c).
– **Phase Coding**: It encodes data based on a global oscillator as the oscillating signals in the brain (e.g., delta: 1-3 Hz, theta: 4-8 Hz, alpha: ∼10 Hz, beta: 15-25 Hz, and gamma: 30-100 Hz) are believed to influence the spiking activity [Fri05, Fal19]. To do this, the oscillation is used to determine the timing of spikes. For instance, the spikes for higher-intensity data are distributed at the times when the corresponding amplitude is higher than the spikes for lower-intensity data, as shown in Figure 2.10(d).
– **Rank-Order Coding** [TG98]: It encodes data based on the order of spikes, and the exact timing between spikes is not considered. To do this, a specific data value is converted into a specific order of spikes, thereby requiring a population of neurons to generate the expected order.

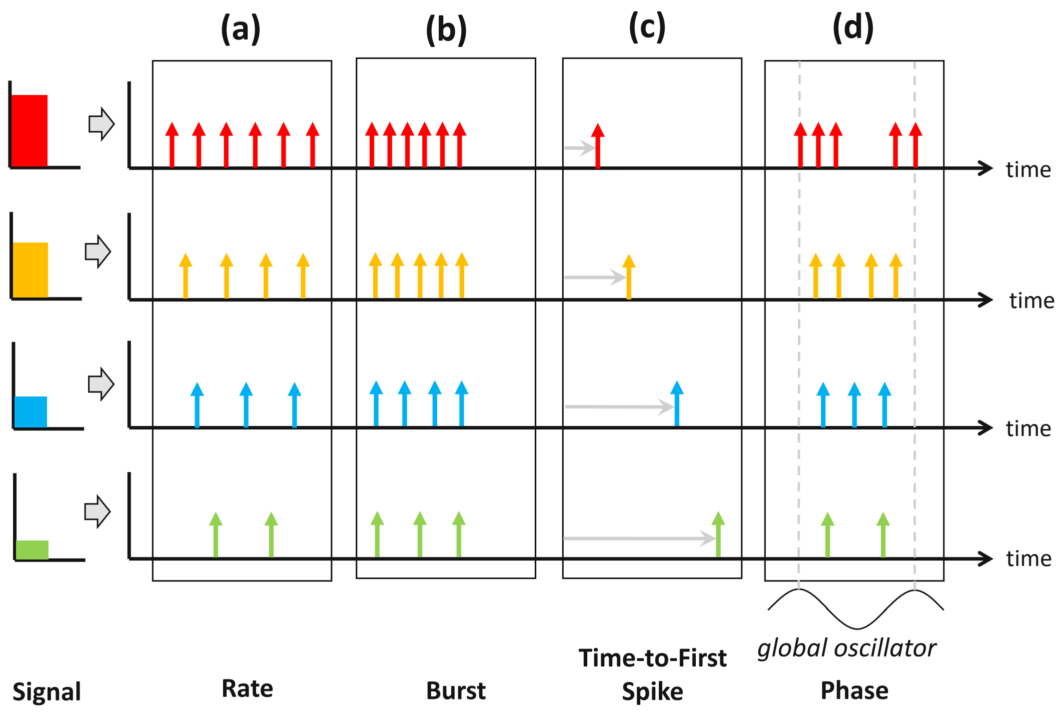


Figure 2.10: Illustration of different neural coding: (a) rate coding, (b) burst coding, (c) time-to-first spike coding, and (d) phase coding (adapted from [PKCY19]).

### 2.3.3 SNN Learning Approaches

In the literature, there are different learning approaches for SNNs, which can be categorized as *supervised learning* and *unsupervised learning* [MGNDM19]. Descriptions of these learning approaches are presented in the following.

- **Supervised Learning**: This approach needs a labeled dataset to train the network. However, the prominent learning methods in the DNN domain (e.g., gradient backpropagation) cannot be directly used in SNNs as its loss function for spikes is not differentiable [RKL$^+$19]. Therefore, researchers have proposed several *supervised learning rules* to overcome such a limitation, such as *SpikeProp* [BKL02], *remote supervised learning (ReSuMe)* [KP06], *Chronotron* [Flo12], *spike pattern association neuron (SPAN)* [MSMK12], *DNN-to-SNN conversion*, and *surrogate gradient learning*. Among them, the most popular and recent ones are described in the following.
  - *DNN-to-SNN Conversion*: It performs network training in the non-spiking domain using the backpropagation, then converts the trained model and input data into SNN domain [RLH$^+$17]. Another technique is through a hybrid conversion, i.e., performing training in the spiking domain after conversion using approximate backpropagation [RSPR20]. Note, this DNN-to-SNN conversion technique can only be used for static datasets as DNNs cannot be directly trained on event-based data, thereby limiting its energy-efficiency gains.
  - *Surrogate Gradient Learning*: Its idea is to adapt the backpropagation concept from the DNN domain to the SNN domain, by approximating the derivative of the loss function of spikes during the backward propagation [NMZ19]. Some related works are the Spike Layer Error Reassignment in Time (SLAYER) [SO18] and Spatio-Temporal Back-Propagation (STBP) [WDL$^+$18]. Further enhancements of this learning approach may enable online training on the neuromorphic hardware, such as the Deep Continuous Local Learning (DECOLLE) [KMN20] due to its localized learning mechanism in synapses.
- **Unsupervised Learning**: This approach can use an unlabeled dataset to train the network, which is very beneficial for enabling energy-efficient smart computing systems due to the following reasons. First, gathering an unlabeled dataset is significantly easier and cheaper, as the costly data labeling process is avoided [RPR19]. Second, unsupervised learning enables continual learning capabilities as the training can be efficiently performed at run time (i.e., online training) using unlabeled data from the environments [PS21b, PS22a, PS23a]. In the literature, several *unsupervised learning rules* have been developed, and they typically have higher biological plausibility than the supervised ones [TGK$^+$19]. Hence, they are known as *bio-plausible learning rules*, whose ideas are described in the following.
  - *Hebbian Rule* [RS97]: It strengthens a synapse if both the connected presynaptic and postsynaptic neurons have high spiking rates at the same time. If it considers a

single spike event, then the weight change depends on the time difference between a presynaptic and postsynaptic spike.

– *Spike-Driven Synaptic Plasticity (SDSP)* [FAB+00, BSF07]: It changes the weight value at the time when a presynaptic spike happens ($t_{pre}$). Weight potentiation (i.e., $\Delta wgh = +a$) is performed if the $v_{mem}$ is higher or equal to the $v_{th}$, and weight depression (i.e., $\Delta wgh = -b$) is performed if the $v_{mem}$ is lower than the $v_{th}$, while considering the concentration of Calcium (Ca) as compared to concentration thresholds $\theta_1$, $\theta_2$, and $\theta_3$. The SDSP rule can be formulated as Equation 2.12.

$$\Delta wgh = \begin{cases} +a & \text{if} \quad v_{mem}(t_{pre}) \geq v_{th} \quad \text{and} \quad \theta_1 \leq \text{Ca}(t_{pre}) < \theta_3 \\ -b & \text{if} \quad v_{mem}(t_{pre}) < v_{th} \quad \text{and} \quad \theta_1 \leq \text{Ca}(t_{pre}) < \theta_2 \end{cases} \tag{2.12}$$

– *Spike-Time-Dependent Plasticity (STDP)* [RAIAS+14]: It updates the weight value based on the temporal correlation between the presynaptic and postsynaptic spikes. STDP learning rule has two variants, i.e., *pair-based STDP* and *triplet-based STDP*. The pair-based STDP depends on a pair of the presynaptic and postsynaptic spikes, while the triplet-based STDP depends on the triplet combinations of spikes. Due to its simplicity and bio-plausibility, the pair-based STDP is more desired and have been widely used in the SNN community [PP18, TGK+19]. The pair-based STDP can be formulated as Equation 2.13-2.14. Here, $A^+$ and $A^-$ denote the amplitude parameters for potentiation and depression, respectively. $\tau_+$ and $\tau_-$ denote the time constants for potentiation and depression, respectively. Meanwhile, the time difference between the presynaptic spike time ($t_{pre}$) and postsynaptic spike time ($t_{post}$) is denoted as $\Delta t$.

$$\Delta wgh = \begin{cases} A^+ \ e^{\frac{-\Delta t}{\tau_+}} & \text{if} \quad \Delta t > 0 \\ -A^- \ e^{\frac{\Delta t}{\tau_-}} & \text{if} \quad \Delta t > 0 \end{cases} \tag{2.13}$$

$$\Delta t = t_{post} - t_{pre} \tag{2.14}$$

The $\Delta wgh$ in pair-based STDP is often computed using synaptic traces to improve the computation speed (e.g., in simulation) [MAD07], as shown in Equation 2.15. $\eta_{pre}$ and $\eta_{post}$ denote the learning rate for the presynaptic and postsynaptic spike event, respectively. $x_{pre}$ and $x_{post}$ denote the traces for the presynaptic and postsynaptic spike event, respectively. When a spike occurs, the corresponding trace is set to 1, otherwise the trace decreases, as shown in Figure 2.11. Meanwhile, $wgh$ denotes the current weight value, $wgh_{max}$ denotes the maximum weight value, and $\mu$ denotes the weight dependence factor. *In this thesis, most of the case studies consider the pair-based STDP learning rule.*

$$\Delta wgh = \begin{cases} \eta_{pre} \ x_{post} \ wgh^{\mu} & \text{on presynaptic spike} \\ -\eta_{post} \ x_{pre} \ (wgh_{max} - wgh)^{\mu} & \text{on postynaptic spike} \end{cases} \tag{2.15}$$

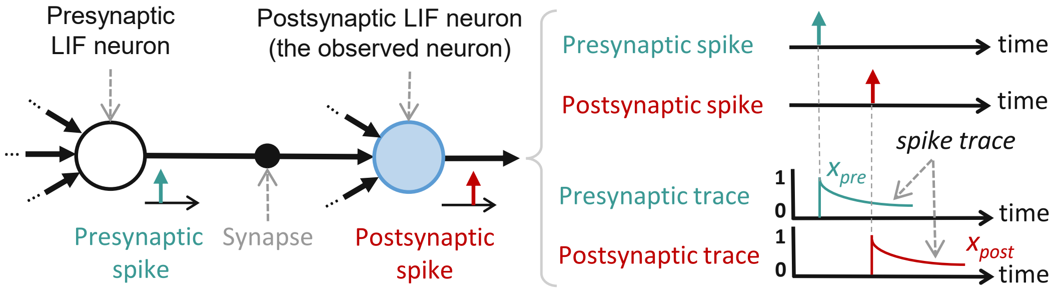


Figure 2.11: Illustration of the dynamics of spike traces for the presynaptic spike event $x_{pre}$ and the postsynaptic spike event $x_{post}$ (adapted from [PS20]).

### 2.3.4 SNN Architectures

SNN architecture or topology is defined as the structure of how spiking neurons are connected to each other through synapses. In general, the spiking neurons are arranged in the form of layers, including an input layer, hidden layer(s), and an output layer. Several SNN architectures have been explored, as described in the following.

- **Feed-Forward Networks**: In the feed-forward architecture, multiple layers are ordered subsequently. Neurons in layer $l$ are connected to the neurons in the subsequent layer (layer $l$+1), and FC layers are typically employed [TGK+19]. Figure 2.12(a) shows the most popular feed-forward architecture for SNNs (i.e., FC-based SNNs), which was introduced by the work of [DC15].

  This architecture consists of input, excitatory, and inhibitory layers. The input layer contains an input image, where every pixel is connected to all excitatory neurons. Each excitatory neuron has to recognize a class in the dataset, and the connecting synapses from the same neuron have to learn the features of the corresponding class. The excitatory neurons are connected to inhibitory neurons in a one-to-one connection. Each spike from an excitatory neuron triggers the corresponding inhibitory neuron to generate a spike that will be delivered to all excitatory neurons, except for the one from which the inhibitory neuron receives a connection. This inhibition provides competition among excitatory neurons. Here, a winner-takes-all (WTA) mechanism is employed to determine the classification. Furthermore, to prevent a neuron from dominating the spiking activity, the neurons' membrane threshold potential ($v_{th}$) is usually defined by $v_{th} + \theta$. The $\theta$ denotes the adaptation potential which is increased each time the corresponding neuron generates a spike, and otherwise, it decays with a rate of $\theta_{decay}$. *In this thesis, most of the case studies consider a similar architecture, as it has demonstrated high accuracy when employing the bio-plausible learning rules under unsupervised learning settings*, thereby making it suitable for energy-efficient SNN training and inference on neuromorphic hardware.

- **Convolutional Networks**: In the convolutional architecture, multiple convolutional layers with specific sizes are arranged in a specific order. The convolutional-based SNN architectures can be developed through two main approaches. First, the development

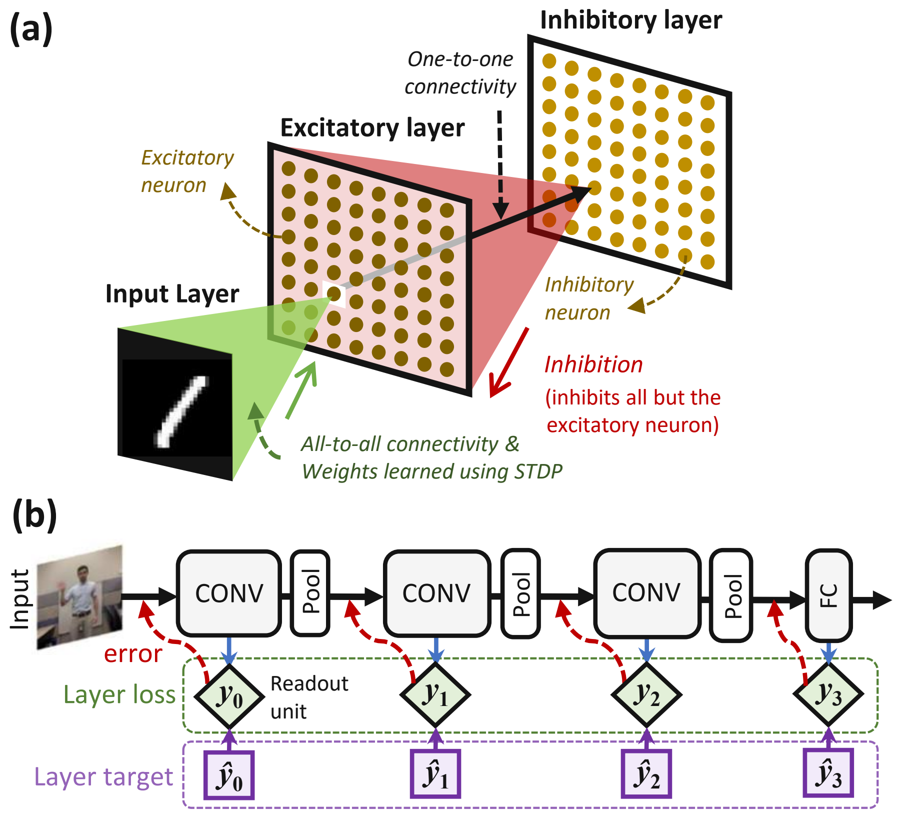


Figure 2.12: (a) An SNN architecture with unsupervised-based local STDP learning rule (adapted from [DC15]). (b) The multi-layer SNN architecture with supervised-based local learning called DECOLLE (adapted from [KMN20]).

is performed using DNN-to-SNN conversion. Here, the architectures follow the given DNNs (e.g., AlexNet and VGG-16) [RLH$^+$17]. As consequence, the benefits of SNNs are limited only to the inference phase, as the training is performed in the DNN domain. Second, the development is performed directly in the SNN domain [CCK15, KMN20]. Among the existing convolutional-based SNNs, a network shown in Figure 2.12(b) can achieve promising accuracy by employing the supervised Deep Continuous Local Learning (DECOLLE) [KMN20], which makes its architecture more suitable for training directly on neuromorphic hardware than other convolutional-based SNNs.

The DECOLLE network consists of 3 convolutional layers and 1 FC layer, as shown in Figure 2.12(b). Each network layer is trained using a surrogate gradient for minimizing the local (layer-wise) loss function, so that the readout unit can produce the targeted output ($\hat{y}$). The difference between the readout output ($y$) and the target ($\hat{y}$) denotes the error that is used to train the weights (red-dashed line). In this manner, the loss function minimization can be performed directly in the spiking environment. The dynamics of each layer are based on the current-based LIF neuron [SJ20], and can be expressed as Equation 2.16. $V_i^l[n]$ denotes the membrane potential of neuron-$i$ in layer-$l$ at timestep-$n$, while $w_{ij}$ denotes the weight between the pre-synaptic neuron-$j$ and the post-synaptic neuron-$i$. A spike $S_i^l[n]$ is emitted at timestep-$n$ if $V_i^l[n]$ reaches the threshold ($V_{th}$) through the $\Theta$ function, where $\Theta(x) = 1$ if $x \geq 0$, and otherwise 0. $P$ and $Q$ denote the traces of the membrane and the current-based synapse respectively, while $R$ denotes the refractory state and $\rho$ is the inhibition weight. $\alpha = exp(-\frac{\Delta t}{\tau_{mem}})$, $\beta = exp(-\frac{\Delta t}{\tau_{syn}})$, and $\gamma = exp(-\frac{\Delta t}{\tau_{ref}})$ denote the decay of the $V$, $Q$, and $R$, respectively [SJ20]. *In this thesis, we consider the DECOLLE network for some case studies under supervised learning scenarios.*

$$\begin{aligned}
V_i^l[n] &= \sum_j w_{ij}^l P_j^l[n] - \rho R_i^l[n] \\
S_i^l[n] &= \Theta(V_i^l[n] - V_{th}) \\
P_j^l[n+1] &= \alpha P_j^l[n] + Q_j^l[n] \\
Q_j^l[n+1] &= \beta Q_j^l[n] + S_j^{l-1}[n] \\
R_i^l[n+1] &= \gamma R_i^l[n] + S_i^l[n]
\end{aligned} \tag{2.16}$$

- **Recurrent Networks**: This architecture has some cycles of connections in the network. Reservoir computing is an example of recurrent networks. Reservoir computing employs a reservoir layer that contains a population of randomly connected neurons [SVVC07].

### 2.3.5 SNN Training and Inference

SNN training and inference process depends on the learning approach since the processing will follow a specific learning rule, as described in Section 2.3.3. For instance, if we consider a widely used FC-based SNN model from [DC15] with rate coding and STDP learning rule, then the training and inference phases are performed under the unsupervised learning settings (see Figure 2.13).

*In the training phase*, each sample of a dataset is fed into the network. Each data value from the sample (e.g., a pixel value of an image) is mapped to a specific neuron in the input layer. This value is converted into a spike train based on the rate coding. Each spike train travels to the connected excitatory neurons. Then, neuronal dynamics in each excitatory neuron decide if the neuron will generate an excitatory spike train. Afterward, the timing correlation between the input spikes and the excitatory spikes is leveraged to train the network, i.e., update the weight value of the connecting synapse using the STDP learning rule. The excitatory neurons are categorized based on their highest response to different classes over the training phase due to the rate coding, hence each neuron is assigned to a specific class (i.e., class-assigned neuron). Meanwhile, *in the inference phase*, the weight values are kept unchanged. The process of data conversion and spike propagation is the same as the training phase. The excitatory spike trains are leveraged to indicate the classification. Here, the response of the class-assigned neurons is used to measure the accuracy.

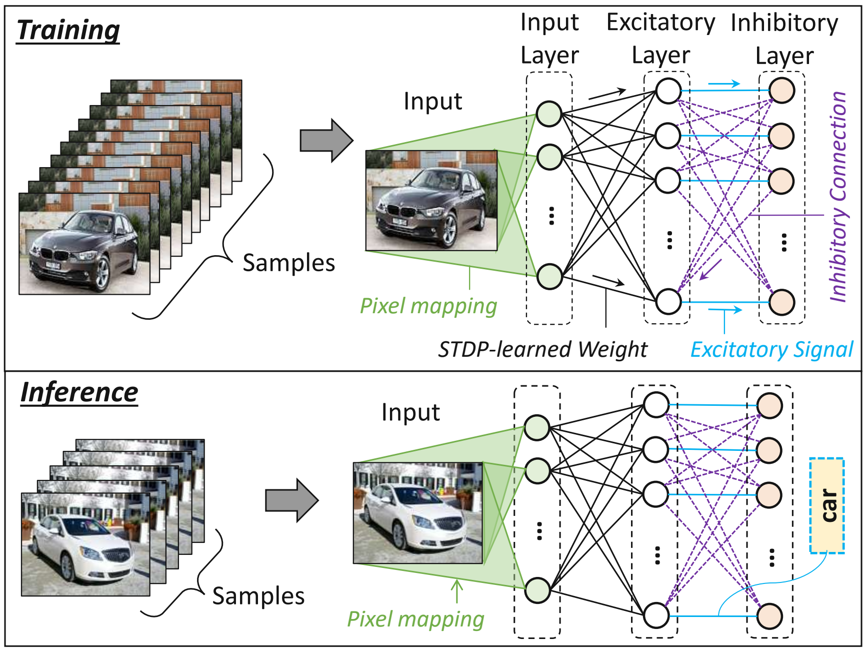


Figure 2.13: An illustration of an FC-based SNN model during (a) training phase, and (b) inference phase.

### 2.3.6 Neuromorphic Hardware Accelerators

SNN processing is highly inspired by the biological brain due to the employment of spike-encoded information, spike-based operations, and bio-plausible learning rules. Therefore, SNN processing requires a specialized computing approach (i.e., *neuromorphic computing*) to maximize the performance and energy efficiency of the SNN training and/or inference phase. The prominent solution is by employing neuromorphic HW accelerators [SPP+17, BDFZ22, LDT+23]. In the literature, there are three main categories of neuromorphic accelerator designs: (1) *large-scale multi-core designs*, (2) *digital single-core designs*, and (3) *mixed-signal single-core designs*. The representative designs for different categories are presented in Table 2.3.

Among the above-mentioned neuromorphic accelerators, the most popular designs are described in the following.

- **HICANN** [SBG+10]: HICANN stands for *High Input Count Analog Neural Network*, and it is part of the BrainScaleS project. This chip contains 512 neurons and 112K synapses in crossbar style. It is implemented in 180 nm CMOS technology with a mixed-signal design, which results in a chip with 49 mm$^2$ of area. It supports both, the inference process and the on-chip learning with the STDP rule.

- **SpiNNaker** [PPG+13]: It employs microprocessors to enable higher flexibility than silicon neurons. It consists of 18 ARM microprocessors with toroidal connection

Table 2.3: A representative list of state-of-the-art neuromorphic accelerator designs.

| **Multi-core Design** (Digital or Mixed-signal) | **Single-core Design** | |
|---|---|---|
| | **Digital** | **Mixed-signal** |
| SpiNNaker [PPG+13] $^{(D,\ OL,\ m)}$ | [SBL+11] $^{(D,\ X,\ OL,\ a)}$ | HICANN [SBG+10] $^{(M,\ X,\ OL,\ a)}$ |
| IFAT [PHY+14] $^{(M,\ a)}$ | ODIN [FLLB19] $^{(D,\ X,\ OL,\ a)}$ | [BNH+13] $^{(M,\ X,\ OL,\ a)}$ |
| TrueNorth [ASC+15] $^{(D,\ X,\ a)}$ | [PLJ20] $^{(D,\ OL,\ a)}$ | Neurogrid [BGM+14] $^{M,\ a)}$ |
| DYNAPs [MQSI17] $^{(M,\ a)}$ | $\mu$Brain [SSYC21] $^{(D,\ a)}$ | ROLLS [QMC+15] $^{(M,\ X,\ OL,\ a)}$ |
| Loihi [DSL+18] $^{(D,\ X,\ OL,\ a)}$ | IMPULSE [AAK+21] $^{(D,\ X,\ a)}$ | [MPN+16] $^{(M,\ X,\ OL,\ a)}$ |
| Tianjic [PDS+19] $^{(D,\ X,\ a)}$ | [GWG+22] $^{(D,\ X,\ f)}$ | MNIFAT [MET+17] $^{(M,\ X,\ a)}$ |
| MorphIC [FLB19] $^{(D,\ X,\ OL,\ a)}$ | ESSA [KCW+22] $^{(D,\ X,\ f)}$ | Braindrop [NFB+19] $^{(M,\ a)}$ |
| Novena [NPL+20] $^{(D,\ X,\ a)}$ | Cerebron [CGF22] $^{(D,\ X,\ f)}$ | [WKE+20] $^{(M,\ X,\ a)}$ |
| | OpenSpike [MGE23] $^{(D,\ a)}$ | |

*D*: Full digital design; *M*: Mixed-signal design.
*X*: Crossbar-based design; *OL*: The design supports on-chip learning.
*a*: ASIC; *f*: FPGA; *m*: General-purpose microprocessor.

topology. It is implemented in 130 nm CMOS technology with full digital design, which results in a chip with 88 mm$^2$ of area. It supports both, the inference process and the on-chip learning with flexible rules as it employs general-purpose microprocessors.

- **Neurogrid** [BGM+14]: It consists of 16 interconnected chips (i.e., Neurocores), and each Neurocore chip has a 256×256 grid of compute units. It is implemented in 18 nm CMOS technology with a mixed-signal design, which results in a chip with 149 mm$^2$ of area. It only supports the inference phase and cannot perform on-chip learning.

- **ROLLS** [QMC+15]: ROLLS stands for *Reconfigurable On-Line Learning Spiking neuromorphic processor*. It consists of 256 neurons and 128 K synapses. It is implemented in 180 nm CMOS technology in a mixed-signal design, which results in a chip with 44 mm$^2$ of area. It supports both inference and on-chip learning with the SDSP rule.

- **TrueNorth** [ASC+15]: TrueNorth chip is developed under the SyNAPSE project. It consists of 4096 neuromorphic cores, each containing 256 LIF neurons and a 256×256 crossbar of synapse connections. It only supports the inference phase, hence it cannot perform on-chip learning. It is implemented in 28 nm CMOS technology with full digital design, which results in a chip with 413 mm$^2$ of area.

- **Loihi** [DSL+18]: It is a neuromorphic manycore processor with on-chip learning from Intel. It consists of 128 cores, including around 130K neurons and around 130M synapses. Loihi is implemented in 14 nm CMOS technology with full digital design, which results in a chip with 60 mm$^2$ of area. It supports both the inference process and the on-chip learning with flexible rules, such as the STDP rule.

- **Tianjic** [PDS+19]: This chip has 156 cores, each of which consists of the dendrite, synapse array, soma, axon, and router units, with around 22 KB of SRAM for weight memory. It is implemented in 28 nm CMOS technology, which results in a chip with 14.4 mm$^2$ of area. It only supports inference and cannot perform on-chip learning.

- **ODIN** [FLLB19]: This chip has a single core with 256 neurons and 256×256 synapses, with 4 KB neuron memory and 32 KB synapse memory. It is implemented in 28 nm CMOS technology with full digital design, which results in a chip with 0.086 mm$^2$ of area. It supports the inference process and the on-chip learning with the SDSP rule.

The above literature highlights the design characteristics of neuromorphic accelerators required for achieving high-performance and ultra-low-power/energy-efficient AI systems, as follows. First, a single-core design with a crossbar-based compute engine is favored to achieve good trade-offs between performance (i.e., speed) and efficiency (e.g., area, power/energy). Second, recent works consider the digital design approach to ensure the reliability of computation. Furthermore, neuromorphic accelerators should have on-chip learning capabilities to enable online learning for adaptive computing systems. Therefore, the representative architecture of neuromorphic accelerators may follow designs from [FLLB19, GWG$^+$22, KCW$^+$22, CGF22], which can be illustrated as Figure 2.14-2.15. *In this thesis, most of the case studies consider a similar design, i.e., a digital single-core neuromorphic accelerator with a crossbar-based compute engine and an on-chip learning unit.* Note, we refer the compute engine of neuromorphic accelerators to as the "SNN compute engine" for conciseness.

Figure 2.16(a) presents the overview of the dataflow for mapping weights on the synapses, feeding spike trains, and performing computations on the compute engine. If we consider an array of synapses with $M \times N$ size (with $M$ and $N$ denoting the number of rows and columns of synapses, respectively), then only a set of $M \times N$ weights from the weight buffer can be mapped on the compute engine at one time. The dataflow maps a specific set of filters to a column of the compute engine at one time, hence only $M$ weights for the same neuron are mapped. Once the weights are mapped to the synapses, they are kept stationary. Then, the input spike trains are fed from the left side of the compute engine. In each synapse, each input signal (i.e., spike or no-spike) determines if the weight value in the local register is propagated to the adjacent synapse and accumulated for updating the neuronal dynamics. To ensure that synapses can operate in lockstep to generate the correct accumulated weight value, the input spike trains must be arranged properly, as shown in Figure 2.16(b).

In the first clock cycle, synapses $S_{1,1}$ receives an input signal (i.e., spike or no-spike) from the first input set. If a spike presents, $S_{1,1}$ propagates its weight value in the local register to the next synapse in the same column ($S_{2,1}$). Otherwise, $S_{1,1}$ propagates zero. In the same clock cycle, all other synapses from the same row also receive the same input signal. In the second clock cycle, synapse $S_{2,1}$ receives the second input signal (i.e., spike or no-spike) from the first input set, propagates the weight value based on the presence of the input spike, adds the weight value from $PE_{1,1}$, and then stores the accumulated weight value in its local accumulation register. Furthermore, in the same clock cycle, synapses at the first row also receive the first input signal (i.e., spike or no-spike) from the second input set. Then, these synapses propagate their respective stored values (i.e., weight or zero based on the input signal) to the synapses in the second row. This

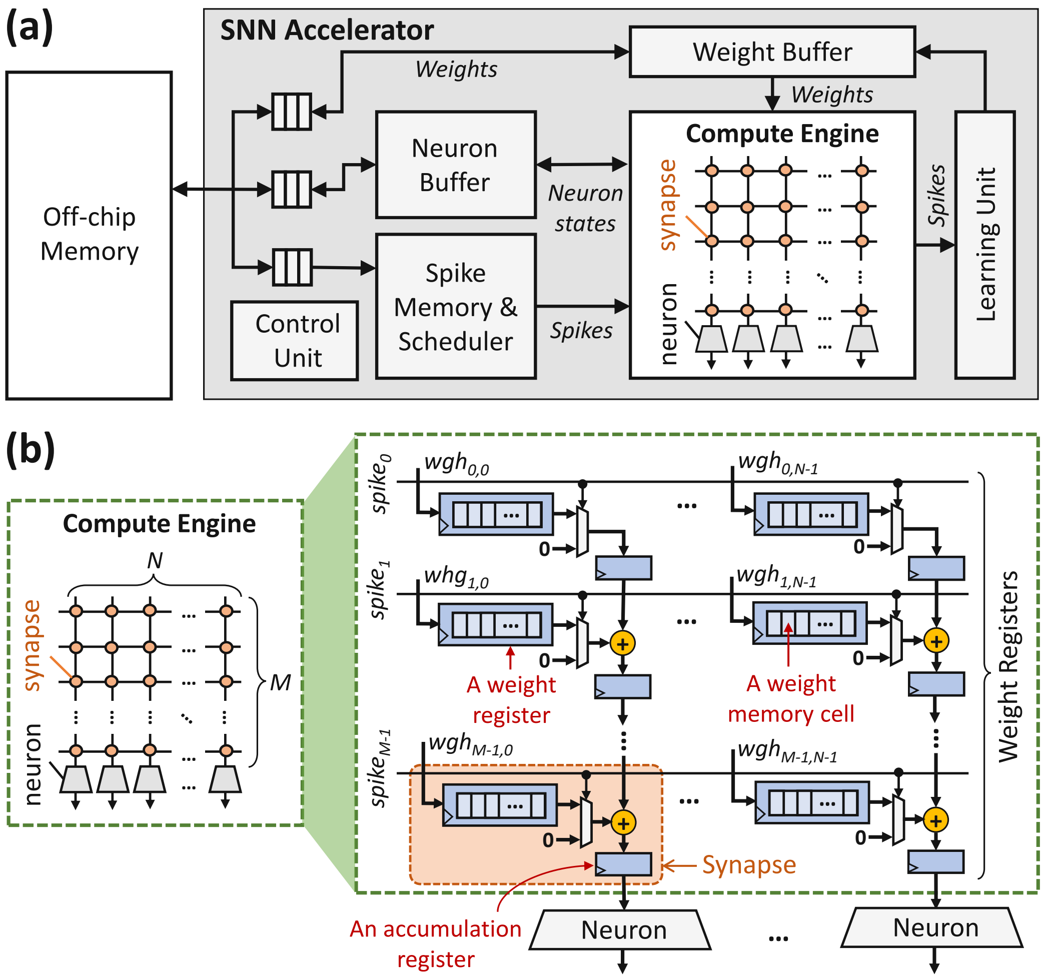


Figure 2.14: (a) An overview of the neuromorphic accelerator. (b) A detailed view of the crossbar-based compute engine architecture.

process goes forward until the last row of synapses. In this manner, the compute engine generates its first accumulated weight value after $M$ clock cylces, and can generate up to $N$ accumulated weight values simultaneously. These values are then processed in neurons to update their membrane potentials and generate output spikes. If the number of weights for the same neuron is more than the number of rows in the array ($M$), then the filter needs to be divided into blocks, and each block has a maximum of $M$ number of weights. Meanwhile, if the number of neurons in a layer is more than the number of columns in the array ($N$), then the filters need to be divided into sets, and each set has a maximum of $N$ number of filters.

**Neuromorphic Accelerators with Conventional and Emerging Technologies:** In conventional computing paradigm, neuromorphic accelerators employs the von-Neumann architecture with CMOS technology, which leads to frequent data movements between

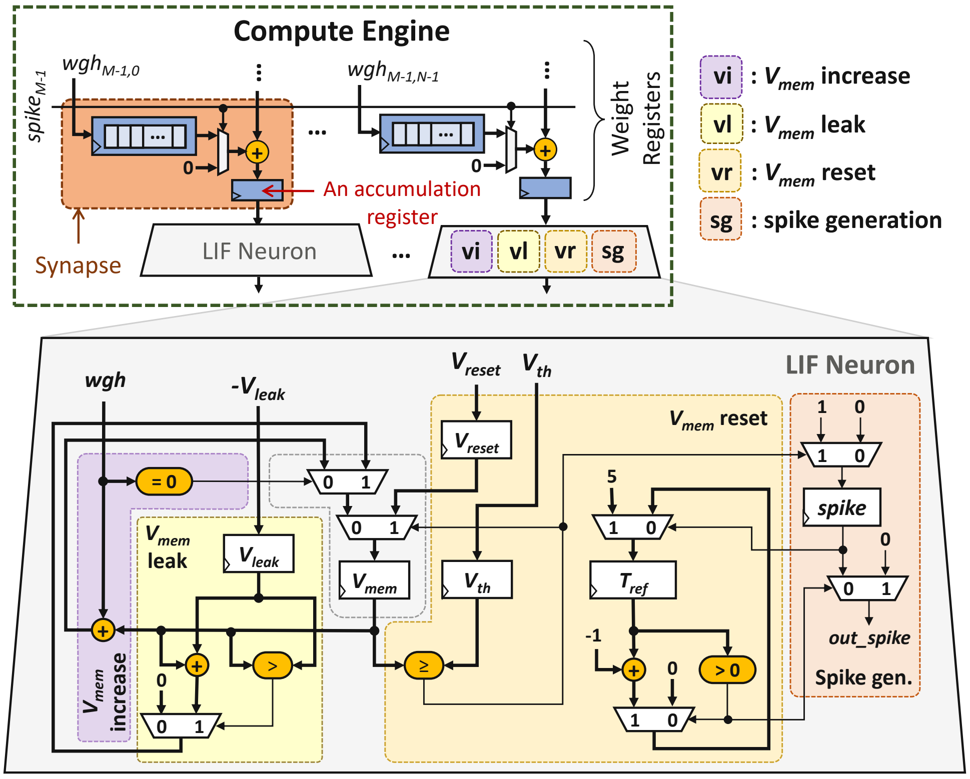
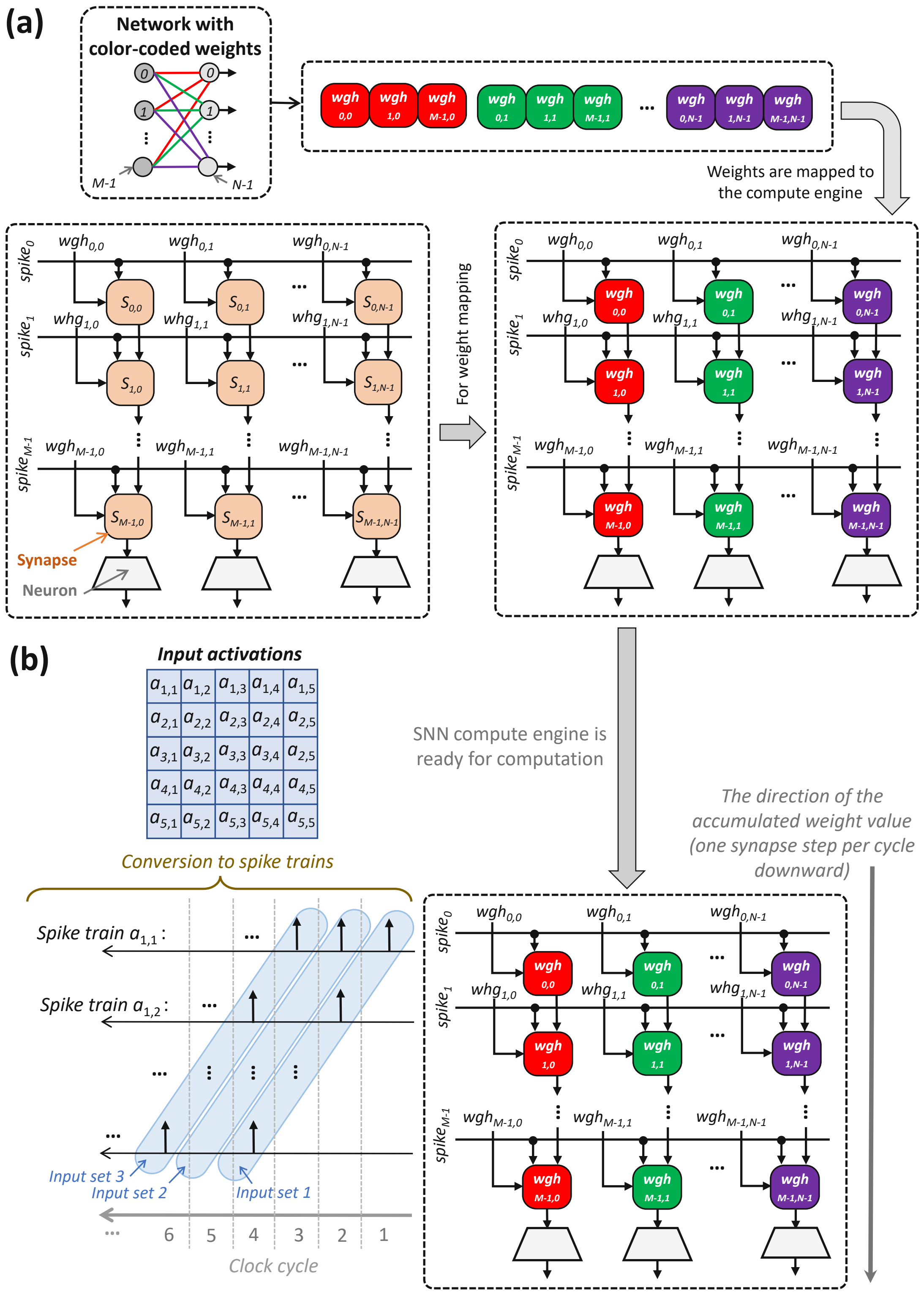


Figure 2.15: A detailed view of the neuron hardware (i.e., LIF neuron).

compute engines and memory modules. To minimize data movements, recent works developed neuromorphic accelerators with non-von-Neumann architecture and emerging technologies (e.g., RRAM, MRAM, and CNTFET). However, these emerging NVM devices (e.g., RRAM and MRAM) still face reliability issues, indicating that they are not yet mature and still require further studies [AZH+23]. Moreover, the conventional computing with off-chip DRAM and on-chip SRAM structure have been widely used in real-world applications. Therefore, *in this thesis, we also focus on the conventional computing paradigm with CMOS technology for neuromorphic accelerators to provide immediate impact in both academia and industry.*

### 2.3.7 Techniques for Improving the Energy Efficiency of SNNs

Recent studies observed that memory accesses dominate the total energy of SNN systems [KSVR19], mainly due to a high number of DRAM accesses as well as complex and costly DRAM operations (whose concept is discussed in Section 2.6). Therefore, optimizing memory access is the key to improving these systems' energy efficiency. At the software level, the existing works compressed the SNN model size with an expectation of reduced memory access requirements through pruning [RPR19, EBDB20, GFY+20], and quantization [RPR19] whose concept is discussed in Section 2.5. Other works employed approximation to reduce the required neural operations [SVR17] and data bundling



(a)
Network with color-coded weights
Weights are mapped to the compute engine
For weight mapping
Synapse
Neuron
(b)
Input activations
Conversion to spike trains
Spike train $a_{1,1}$:
Spike train $a_{1,2}$:
Input set 3
Input set 2
Input set 1
Clock cycle
SNN compute engine is ready for computation
The direction of the accumulated weight value (one synapse step per cycle downward)

Figure 2.16: Overview of the dataflow for (a) mapping weight to the compute engine, and (b) feeding the spike trains to the compute engine and performing spike-based computation.

to reduce memory accesses [KSVR19]. At the hardware level, SNN accelerators have been proposed to improve performance efficiency, such as TrueNorth [ASC+15], Loihi [DSL+18], Tianjic [PDS+19], and ODIN [FLLB19]. Table 2.4 presents a summary of the techniques for improving the energy efficiency of the SNN systems. However, there are several limitations to the existing works as discussed in the following.

**Limitations of the State-of-the-Art Works:**

1. *Optimization of the memory requirement of SNNs:* SNN systems with unsupervised learning typically employed a pair of excitatory neurons and inhibitory neurons to support the learning process [DC15, S+17, HSS+18, SPH+19, HSS+19]. Since the inhibitory neurons have different functionality from the excitatory ones, their parameters for connections and neurons are also different from the excitatory ones, thereby occupying a significant amount of memory footprint and incurring high memory access energy. However, this aspect is not optimized by state-of-the-art works.

2. *Hardware-level optimization:* Previous works employed software-level optimization techniques to exploit the error-resilient nature of SNNs to improve energy efficiency, such as pruning [RPR19, EBDB20, GFY+20], quantization [RPR19], and approximate operations [SVR17]. Therefore, they do not consider hardware-level optimization (e.g., employment of approximate hardware units) that can substantially reduce the operational power/energy of the SNN systems.

## 2.4 Reliability Threats in NN-based Computing Systems

### 2.4.1 Overview

The advanced NNs (DNNs and SNNs) require high resource and power/energy budgets due to their huge memory and computing requirements. Therefore, specialized HW accelerators are typically employed to significantly improve the efficiency of NN-based systems. However, these accelerators are still vulnerable to HW-induced reliability threats, such as *approximation errors*, *permanent faults*, and *transient faults*. These challenges are associated with the impact of HW-level optimization (e.g., approximation), limitations of chip fabrication process, and interactions with high-energy particles. A brief description of HW-induced reliability threats is provided below.

- **Approximation Errors** are errors that occur due to approximation techniques that trade the quality (e.g., computing accuracy) for better efficiency of computing systems. These errors can occur in (1) *approximate functional units* such as adders and multipliers [GMP+11, KGE11, REHS+16] due to logic simplification or voltage scaling [RBKS17, VCC+13], and (2) *approximate memories* such as DRAM [KOY+19, CYG+17b] and on-chip buffer [PHS21a] due to voltage and access latency scaling.

Table 2.4: An overview of state-of-the-art works for improving the energy efficiency of the SNN systems [PS20].

| Abstraction Layer | Technique | Related Work | Brief Description | Benefits | Cost / Weaknesses |
|---|---|---|---|---|---|
| Software | Pruning | [RPR19] [EBDB20] [GFY+20] | It removes redundant parameters in a SNN based on their significance, thereby reducing the SNN model size (i.e., memory footprint) and the number of SNN computations. | • Less memory footprint<br>• Less number of computations | • Accuracy loss<br>• Training cost<br>• Data encoding |
| | Quantization | [RPR19] [Wan19] | It reduces the precision of SNN parameters and intermediate outputs by converting them from floating-point to fixed-point format. | • Less memory footprint<br>• Simpler hardware modules | • Accuracy loss<br>• Training cost |
| | Functional Approximations | [SVR17] | It approximates the functionality of SNN components (e.g., neurons) to reduce the required computations. | • Less memory footprint<br>• Simpler hardware modules | • Accuracy loss |
| | Spike Bundling | [KSVR19] | It bundles a set of spikes that occur close in time into a single synaptic spike using spike bundling units (i.e., encoder and decoder). | • Less memory accesses | • Bundling cost |
| Hardware | Hardware Acceleration | [ASC+15] [RVG+17] [SVR17] [DSL+18] [FLLB19] [FLB19] | It expedites the SNN processing on hardware accelerator. Some support inference only, while others support training and inference. | • Faster training and/or inference time<br>• Higher efficiency | • Design time |

- **Permanent Faults** are faults that remain for indefinite periods until corrective action is performed [Muk11]. Permanent faults can come from different sources, as briefly described in the following.

  1. *Chip Fabrication Process:* Fabricating chips with millions-to-billions of nano-scale transistors with 100% correct functionality is difficult, and even worsen due to the aggressive technology scaling [ZGBG18, HKP$^+$18, HKP$^+$21]. It can lead to *manufacturing defects* and extreme *process variations* (i.e., deviations in hardware circuits from the expected characteristics [RTGM13]), which cause permanent faults and decrease the yield of the chips.
  2. *Run Time Operation:* Permanent faults can also occur due to device/transistor wear out (*aging*) and damages caused by various physical phenomena, such as Hot Carrier Injection (HCI), Bias Temperature Instability (BTI), Electromigration (EM), and/or Time Dependent Dielectric Breakdown (TDDB) [RFZJ13, WNL16, HKP$^+$18, BBD19, MKK$^+$20, HKP$^+$21].

- **Transient Faults** are faults that occur once and then disappear (i.e., *soft errors*). They are usually caused by high-energy particle strikes on a chip. These particles can be neutrons that come from cosmic radiations or alpha particles from the packaging materials of the chip [Bau05]. These faults manifest as bit flips in the chip, and the impact of these bit flips can propagate to the application layer, thereby resulting in incorrect outputs and affect the accuracy of the system.

### 2.4.2 Techniques for Improving the Fault Tolerance of SNN Systems

*In this thesis, the studies of fault tolerance are focused on the SNN systems*, because (1) SNNs have the potential to achieve more efficient processing than DNNs, (2) SNNs can provide efficient online learning using their unsupervised learning settings to make their systems adapt to diverse operational environments, and (3) the fault tolerance aspects of SNNs have not been explored thoroughly [PHS21a, PHS22b].

To mitigate faults in SNNs, standard fault-tolerance techniques for VLSI circuits such as *Dual Modular Redundancy* (DMR) [VZBT10], *Triple Modular Redundancy* (TMR) [LV62], and *Error Correction Codes* (ECCs) [LM76, CH84, Sze00], might be employed. However, these techniques require redundant hardware and/or executions, thereby incurring high latency and energy overheads. These overheads coupled with the compute- and memory-intensive nature of SNNs make them infeasible for efficient implementation of SNN systems. Therefore, alternate low-cost techniques are required to improve SNNs against HW-induced reliability threats. Toward this, state-of-the-art works have studied fault tolerance for SNNs, which are summarized in Table 2.5 and briefly described as follows.

- **Fault Modeling for SNNs**: Previous works studied different aspects of fault modeling in SNNs, such as identifying a set of possible faults that can affect SNN components (including neurons and synapses) [VDNA19], and investigating the fault modeling for transistor-level neuron HW [ESSP$^+$20].

- **Studies on the Impacts of Faults on Accuracy**: Previous works studied the impacts of bit flips in weights [VMA+20] and synapse failure (i.e., synapse removal) [SPMJ+20] [RLIS21] on the accuracy, considering different fault rates with random distribution.
- **Fault Mitigation**: Previous works studied different fault mitigation techniques. For instance, training with dropouts, neuron saturation detection, and TMR are employed in [SESA+21] as fault-tolerance strategy that corresponds to fault modeling of neuron HW from [ESSP+20]. Another work employed additional components (i.e., astrocyte units) for enhancing the retraining process [RLIS21].

**Limitations of the State-of-the-Art Works:** The above discussion shows that, current state-of-the-art on fault tolerance works for SNNs still focus on studying the fault modeling and the impact of a specific fault on the accuracy. Moreover, their fault mitigation techniques still rely on costly techniques, such as retraining and redundancy. Therefore, *the impact of HW-induced faults on the system-level accuracy considering the underlying SNN neuromorphic architectures, and the respective fault mitigation techniques, are still an unexplored avenue.*

## 2.5 Quantization in Neural Networks

### 2.5.1 Data Representation and Rounding Schemes

The quantization of NNs typically employs a fixed-point format [Kri18]. This fixed-point format is represented as $Qi.f$, which consists of 1 sign bit, $i$ integer bits, and $f$ fractional bits, and follows the 2's complement format [GD03]. If the sign bit is not required, it can be eliminated, and hence this free bit can be used for integer or fractional parts. Given the fixed-point $Qi.f$, the range of representable values is $[-2^i, 2^i - 2^{-f}]$ and the precision is $\epsilon = 2^{-f}$. In the quantization process, a *rounding scheme* is required, and we consider the widely used ones, i.e., truncation, rounding-to-the-nearest, and stochastic [HMLF20, GAGN15].

**Truncation (TR):** This scheme keeps the $f$ bits and discards the other bits from the fractional part. Hence, the output fixed-point for the given real number $x$ and configuration $Qi.f$, is defined as $TR(x, Qi.f) = \lfloor x \rfloor$.

**Rounding-to-the-Nearest (RN):** This scheme rounds the value, that is halfway between two representable values ($\lfloor x \rfloor + \frac{\epsilon}{2}$), by rounding it up. Therefore, the output fixed-point for the given real number $x$ and configuration $Qi.f$, is defined as

$$RN(x, Qi.f) = \begin{cases} \lfloor x \rfloor & \text{if } \lfloor x \rfloor \leq x < \lfloor x \rfloor + \dfrac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon & \text{if } \lfloor x \rfloor + \dfrac{\epsilon}{2} \leq x < \lfloor x \rfloor + \epsilon \end{cases} \tag{2.17}$$

**Stochastic Rounding (SR):** This scheme rounds the value using a non-deterministic approach. Given a random value $P \in [0, 1)$ that is drawn from a uniform random number

Table 2.5: An overview of state-of-the-art techniques for studying and/or improving the fault tolerance of the SNN systems [PHS21a, PHS22b, SLS23].

| Study | Technique / Approach | Related Work | Brief Description | Targeted Faults | Benefits | Cost / Weaknesses |
|---|---|---|---|---|---|---|
| Fault Modeling | Logical Description | [VDNA19] | It describes a set of possible faults in SNNs, covering faults in neurons and synapses. | Permanent and transient faults | A list of possible faults in SNNs | No experiments |
| | Transistor-level Neuron | [ESSP+20] | It studies fault modeling for neuron HW at transistor-level, but only covers Integrate-and-Fire (IF) neuron model. | Permanent faults in neuron HW | Fine-grained fault modeling for neuron HW | • Only for IF neuron<br>• Faulty synapses are not considered<br>• Design time |
| Impact of Faults on Accuracy | Bit-level Fault Injection | [VMA+20] | It studies the impact of bit flips at the random locations of weights, without considering underlying SNN HW architectures. | Permanent faults in weight bits | Fast fault injection | Imprecise fault model |
| | Synapse-level Fault Injection | [SPMJ+20] [RLIS21] | It studies the impact of synapse failure (removal) at the random locations in the given model, without considering underlying SNN HW architectures. | Permanent faults in synapses | Fast fault injection | Imprecise fault model |
| Fault Mitigation | Fault-aware Training | [SESA+21] [SH23] [RLIS21] | It proposes a fault-aware training to make the model better adapt to faults in neuron HW. | Permanent faults in neuron HW | Adaptive to many SNN compute engine | • Faulty synapses are not considered<br>• Retraining cost |
| | Anomaly Detection | [SESA+21] | It detects anomaly that indicates neuron saturation. | Permanent faults in neuron HW | Active fault tolerance at run time | • Only for IF neuron<br>• Faulty synapses are not considered |
| | TMR | [SESA+21] | It employs TMR technique using three identical neurons to vote for deciding the output class. | Permanent faults in neuron HW | Active fault tolerance at run time | • Only for IF neuron<br>• Large area and energy overheads |

generator, the output fixed-point for the real number $x$ and configuration $Qi.f$ is defined as

$$SR(x, Qi.f) = \begin{cases} \lfloor x \rfloor & \text{if } P \geq \dfrac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{if } P < \dfrac{x - \lfloor x \rfloor}{\epsilon} \end{cases} \tag{2.18}$$

### 2.5.2 Quantization Schemes

There are two widely used quantization schemes in the neural network models, i.e., the *Post-Training Quantization*, and the *In-Training Quantization* (or *Quantization-aware Training*) [Kri18], whose key mechanisms are shown in Figure 2.17.

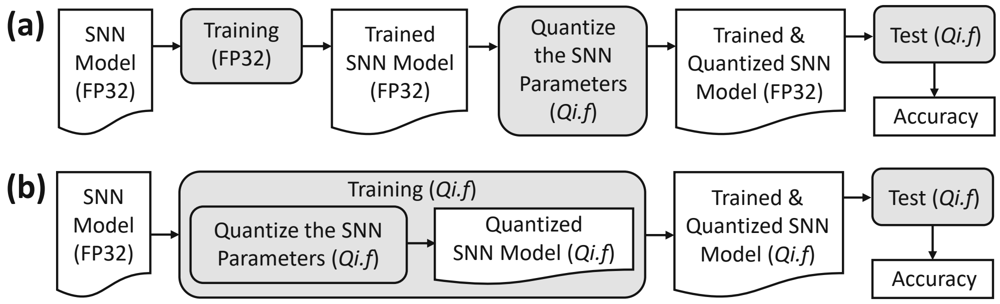

(a)
SNN Model (FP32)
Training (FP32)
Trained SNN Model (FP32)
Quantize the SNN Parameters (Qi.f)
Trained & Quantized SNN Model (FP32)
Test (Qi.f)
Accuracy
(b)
SNN Model (FP32)
Training (Qi.f)
Quantize the SNN Parameters (Qi.f)
Quantized SNN Model (Qi.f)
Trained & Quantized SNN Model (Qi.f)
Test (Qi.f)
Accuracy

Figure 2.17: Overview of (a) the post-training quantization, and (b) the in-training quantization or quantization-aware training.

**Post-Training Quantization (PTQ)** trains an SNN model with a floating-point precision (e.g., FP32) and results in a trained model. Afterward, the quantization is performed on the trained model with the given $Qi.f$ precision, resulting in a quantized model for the inference phase.

**In-Training Quantization (ITQ)** quantizes an SNN model with the given $Qi.f$ precision during the training phase. Therefore, the trained model is already in a quantized form and can be used for the inference phase. The quantization is typically performed using the simulated quantization [Kri18].

*In this thesis, in most of the study cases, we implement the quantization using* ***simulated quantization***, i.e., the network parameters are represented and stored using their low-precision fixed-point values (FxP), and then computed (e.g., through MAC or LIF neuron operations) using floating point arithmetic (FP32) [Kri18, JKC$^+$18b, GKD$^+$, vBKM$^+$22].

## 2.6 DRAM Fundamentals

*Dynamic Random Access Memory* (DRAM) is a specific type of *Random Access Memory* (RAM) that employs a single transistor-capacitor pair for holding a data bit [JWN10]. The capacitor can be charged or discharged, and these two states are used to represent the two values of a single bit: 1 (charged state) and 0 (discharged state). This circuit is dynamic as the electric charge (data) in each capacitor gradually leaks away and will eventually be lost. Therefore, DRAM employs a *refresh mechanism* to periodically rewrite/restore the original data in each capacitor. In this manner, data bits in DRAM can be preserved during the operational time and will disappear when the DRAM is powered off. Therefore, DRAM is also categorized as a *volatile memory*. In modern computing systems, DRAM is employed as the main memory [GLH+19], as shown in Figure 2.18. The main reason is that, DRAM can store large data bits in a smaller area as compared to on-chip *Static RAM* (SRAM), as DRAM employs a smaller number of components (i.e., a transistor and a capacitor) for its storage cell. Furthermore, DRAM can also perform faster data access as compared to commodity storage devices (e.g., hard disk) due to its electric charge/discharge characteristics.

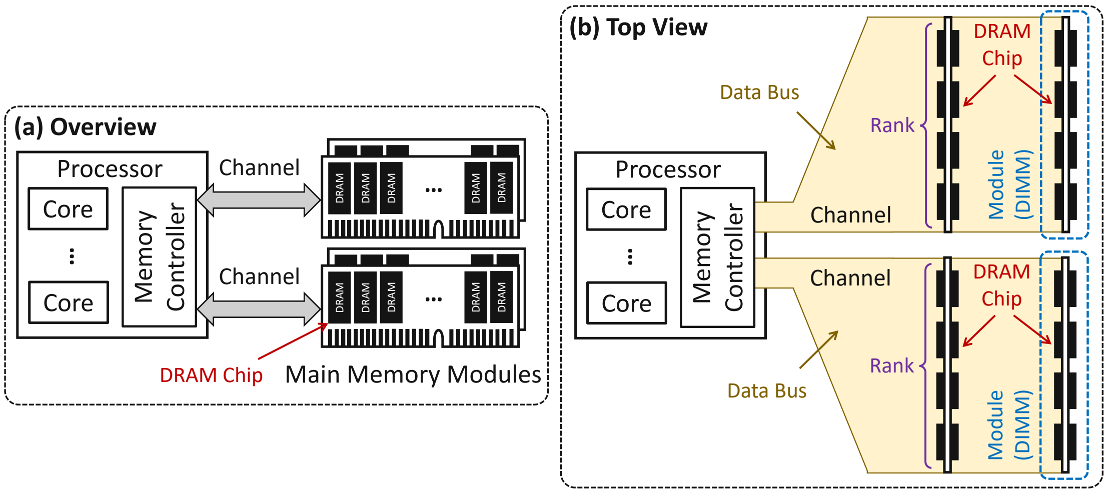


Figure 2.18: The typical computer organization, showing the connections between the processor (including cores and memory controller) and main memory modules: (a) from an overview perspective and (b) from a top-view perspective.

### 2.6.1 DRAM Organization

A DRAM-based memory is organized in a certain hierarchy as seen from a top-down perspective: *channel*, *rank*, *chip*, *bank*, *subarray*, *row*, and *column* [KSL+12, GLH+19]; see an overview in Figure 2.19. The highest level of the hierarchy is a *memory channel* (or simply a *channel*). Each channel has a dedicated bus to the *host* (e.g., processor) and *memory controller*. This channel can connect to one or multiple DRAM module(s), such as *Dual Inline Memory Modules* (DIMMs), as shown in Figure 2.18. Each module typically has several DRAM chips. A group of chips that operate in lockstep is referred to as a DRAM rank. In each DRAM chip, there are several banks, as shown in Figure 2.19(a). These banks are the lowest hierarchy that can operate in parallel, i.e., referred to as *bank-level parallelism* [KPMHB11]. However, these banks also share a single *memory* *bus*, hence the memory controller has to schedule the *memory requests* so that operations in different banks in the same DRAM chip do not interfere with each other [GLH+19]. Furthermore, a DRAM bank is not implemented in a monolithic design (i.e., a large array of cells with a single row buffer). Instead, a bank is implemented in multiple subarrays, and each subarray has its *local row buffer*, as shown in Figure 2.19(b)-(c). Multiple subarrays in a bank share a *global row address decoder* as well as *global bitlines* which connect local row buffers to a *global row buffer* [KSL+12], as shown in Figure 2.19(c).

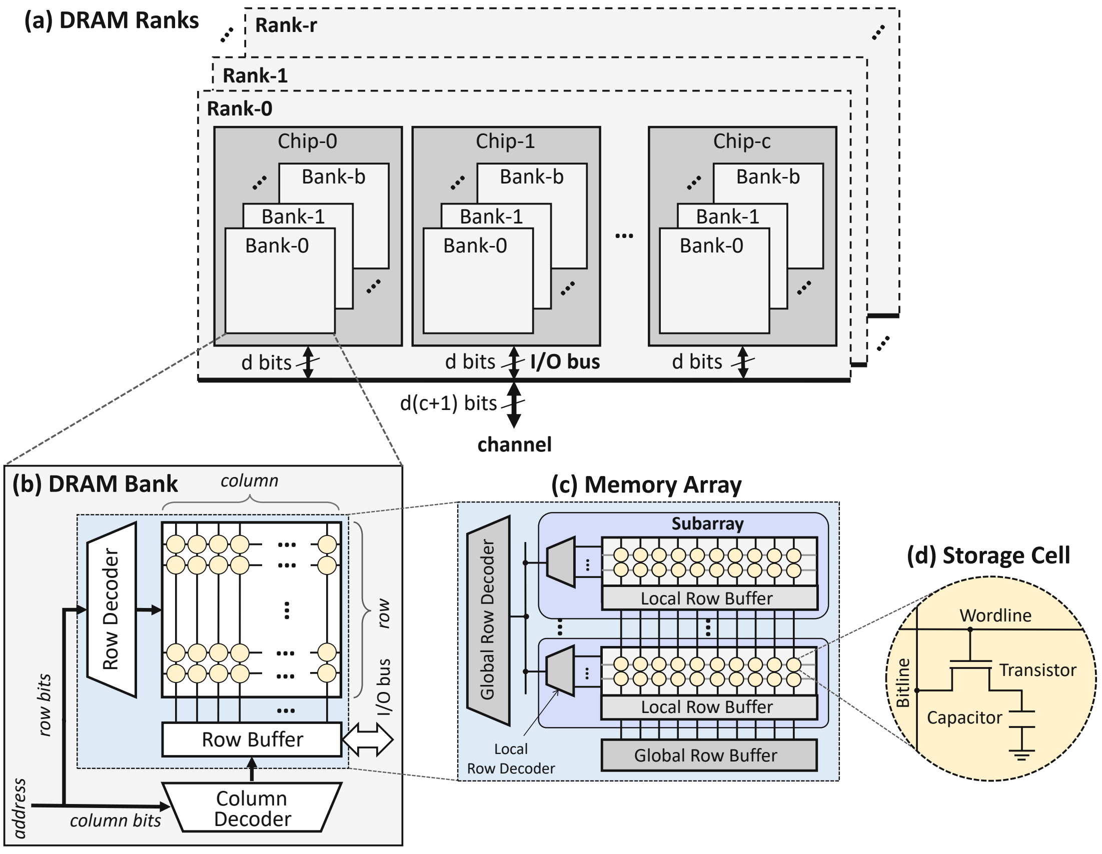


Figure 2.19: DRAM organization. (a) DRAM ranks with an overview of DRAM chips and banks inside a rank. (b) A DRAM bank with an overview of the memory array, row buffer, decoders, and interconnections. (c) A DRAM bank with an overview of the memory subarrays, local and global row buffers, local and global row decoders, and interconnections. (d) A storage cell of DRAM technology.

### 2.6.2 DRAM Operations

#### Basic DRAM Access

To enable a memory request for accessing data at a specific DRAM row and column address, the memory controller issues three commands which trigger a particular sequence of events in the DRAM bank. Following is a brief description of these three commands.

- **Activate (ACT):** It reads of the entire data in the targeted/requested row, then copy the data into the row buffer.
- **Read/Write (RD/WR):** It accesses the requested column from the row buffer, i.e., either reading data for RD command or writing data for WR command.
- **Precharge (PRE):** It deactivates the row buffer, so that the connected banks are ready for receiving another command.

The memory controller sends a request to DRAM through a specific channel, and a targeted DRAM rank will respond. Since there are typically multiple DRAM chips in a rank, then multiple chips in the targeted rank can be accessed in parallel. In each chip, the request is passed to a specific DRAM bank, and then decoded into a DRAM row and column address. The activate (ACT) command triggers a row activation, and data from the requested row are copied to the row buffer (so-called *activated row buffer*). Afterward, either a read (RD) or write (WR) command can be issued by the memory controller to access a certain column in the activated row buffer. There are three different conditions that may be faced in this case: *row buffer hit*, *row buffer miss*, or *row buffer conflict*.

- **Row Buffer Hit:** It happens when the requested row is already activated, hence the data in this row is already copied into the row buffer. In this condition, data in the row buffer can be accessed directly without any new row activation.
- **Row Buffer Miss:** It happens when there is no activated row yet in the row buffer. In this condition, a new row activation is needed and the data in the requested row is copied to the row buffer.
- **Row Buffer Conflict:** It happens when there is an activated row in the row buffer, but its address is not the one that the request is expecting. Therefore, the activated row has to be closed first using the precharge (PRE) command. Afterward, the requested row can be activated using the activation (ACT) command, and the subsequent commands of DRAM access can be issued and executed.

The data accessed from different chips form a DRAM *data word*, which will be transmitted to memory controller, as shown in Figure 2.19(a). For example, if a rank has 8 chips that can be accessed in parallel and each chip provides 8-bit data-per-access, then this rank can provide 64-bit data word-per-access.

### DRAM Timing Constraints

After the memory controller sends a command to DRAM, it has to wait for sufficient amount of time before sending another command. These restrictions are referred to as DRAM *timing constraints*, which are illustrated in Figure 2.20 and described as follows.

- **Row-to-Column Delay** ($t_{RCD}$): It is the time required to copy data in the specified row into the row buffer (i.e., sense amplifier). During this time, the connection of cell to the bitline is established, hence sharing the charge (or the lack of charge) of cell capacitor with the *bitline parasitic capacitor* and perturbing the bitline voltage. Afterward, the sense amplifier senses and amplifies the perturbation until reaching a *threshold state*, where the data is considered to be "copied" to the row buffer.

- **Row Active Time** ($t_{RAS}$): It is the time when a row is active. It includes time for charge sharing, charge sensing, charge amplification, as well as charge restoration to the original value to the cell capacitor. Therefore, $t_{RAS}$ includes $t_{RCD}$.

- **Row Precharge Time** ($t_{RP}$): It is the time required for precharging phase. During this time, the connection of cell to the bitline is terminated, and sense amplifier precharges the bitline voltage by withdrawing charge from (or injecting charge to) bitline parasitic capacitor so that the bitline voltage reaches the quiescent value.

- **Row Cycle Time** ($t_{RC}$): It is the minimum time interval required between two successive ACT commands (ACT $\rightarrow$ ACT) to the same subarray of a bank. Since a new ACT command can only be issued when both the activation phase ($t_{RAS}$) from previous ACT command and precharging phase ($t_{RP}$) to complete, hence the $t_{RC} = t_{RAS} + t_{RP}$.

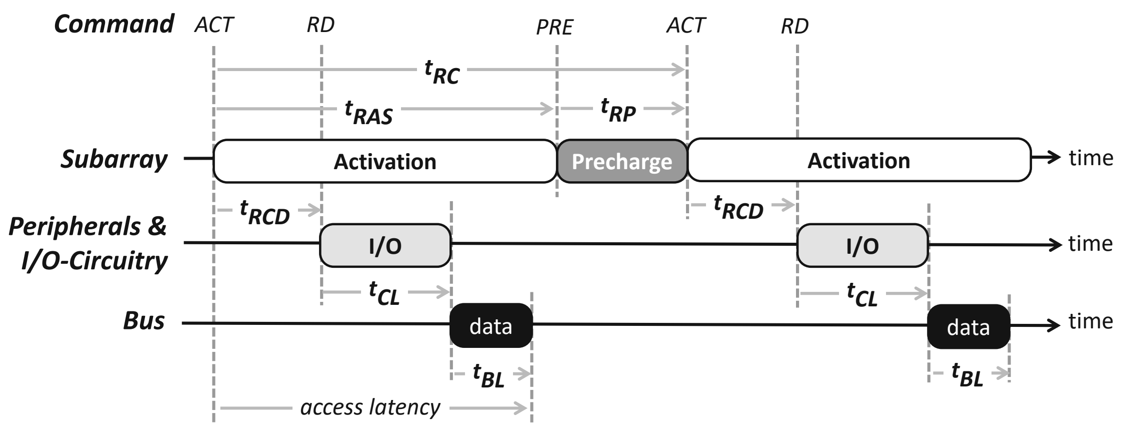


Figure 2.20: Illustration of DRAM accesses with the respective *commands*: ACT, RD/WR, and PRE; *phases*: activation, I/O, and precharge; as well as *timing constraints*: row-to-column delay ($t_{RCD}$), row active time ($t_{RAS}$), row precharge time ($t_{RP}$), column access latency ($t_{CL}$), and data bus latency ($t_{BL}$).

### 2.6.3 Modern DRAM Types

In this subsection, prominent DRAM types are introduced to highlight their specific features, architectural designs, and operations.

### DDR3 and DDR4

In the *Double Data Rate* (DDR) DRAM, a sequence of data is sent on both the positive and negative edges of the DRAM bus clock to double the DRAM data rate. DDR3 is the third generation of *Double Data Rate* (DDR) DRAM, which contains eight DRAM banks in a rank [JED12]. Meanwhile, DDR4 increases the number of banks per rank to 16 and introduces bank groups as a new level of hierarchy in the DRAM [JED17b]. Typically, DDR4 has longer memory access than DDR3 due to the interconnection between the bank groups and I/O of the DRAM chip, but DDR4 provides higher bandwidth than DDR3 due to its significantly higher bus clock frequency.

### Low-Power DDR3 (LPDDR3) and Low-Power DDR4 (LPDDR4)

LPDDR3 is the low-power variant of DDR3 [JED15], while LPDDR4 is the low-power variant of DDR4 [JED17a]. These DRAM types reduce power consumption as compared to their standard-power counterparts through several techniques, like the employment of a lower core voltage, two voltage domains on a single chip, deep power-down modes, reduced chip width, temperature-controlled refresh, and fewer chips-per-module [GLH$^+$19].

### High Bandwidth Memory (HBM)

HBM is a 3D-stacked memory that aims at providing high throughput, where multiple DRAMs are stacked on top of one another. It is designed for computing platforms that require high performance and high bandwidth, like *Graphic Processing Units* (GPUs). To achieve this, each HBM module typically connects multiple memory channels (e.g., 4-8) to a large number of requests in parallel while avoiding the I/O contention [GLH$^+$19].

The above description shows that, *the internal organizations of different DRAM types are the same*, i.e., channel, rank, chip, bank, subarray, row, and column as seen from a top-down perspective. Therefore, data mapping policy in a specific type of DRAM will be applicable to other types of DRAMs [GLH$^+$19].

## 2.6.4 Novel DRAM Architectures

### DRAM with Subarray-level Parallelism (SALP)

In a modern DRAM, each memory request that goes to a DRAM bank, can only access a single subarray at a time. Hence, it limits the potential to further reduce the DRAM access latency and energy. To address this limitation, new DRAM architectures and mechanisms have been proposed to exploit the subarray-level parallelism (SALP) in the same DRAM bank, that are referred to as *SALP-enabled DRAM architectures* or simply *SALP architectures* [KSL$^+$12]. Three types of SALP architectures include SALP-1, SALP-2, and SALP-MASA (see their service time in Figure 2.21), whose key ideas are discussed in the following.

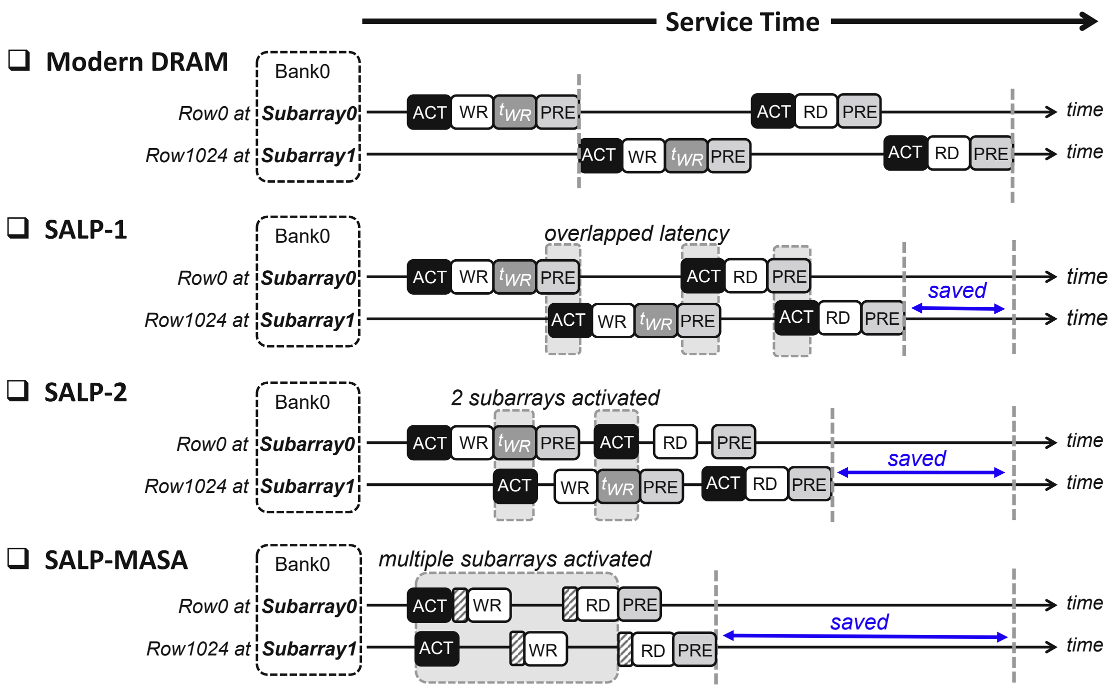


Figure 2.21: Service time for modern DRAMs, SALP-1, SALP-2, and SALP-MASA (adapted from [KSL+12]).

- **SALP-1** reduces the DRAM service time by overlapping the precharge operation of one subarray with the activation operation of another subarray. It can be performed since mostly the precharge and activate operations are local to a subarray. To enable this mechanism, a re-interpretation of the existing timing constraint for precharge operation is required.

- **SALP-2** reduces the DRAM service time by overlapping the *write-recovery latency* ($t_{WR}$) of an active subarray with the activation of another subarray. In this manner, SALP-2 can reduce the DRAM service time even more than SALP-1. To enable the SALP-2 mechanism, additional circuitry to activate two DRAM subarrays at the same time is employed. Note, after the memory controller sends a WR command, the targeted bank requires $t_{WR}$ so that its row buffer can drive the bitlines to new voltages and new values are safely stored in the targeted cells.

- **Multitude of Activated Subarrays (SALP-MASA)** reduces the DRAM service time by overlapping the activation of different subarrays, hence multiple subarrays are activated at the same time. In this manner, SALP-MASA reduces the DRAM service time even more than SALP-1 and SALP-2. To enable the SALP-MASA mechanism, additional circuitry (more complex than the one for SALP-2) to activate multiple subarrays at the same time is employed.

### Tiered-Latency DRAM (TL-DRAM)

Recent study observes that the high latency of DRAM access is a result of the trade-off made by DRAM manufacturers for optimizing the DRAM cost-per-bit [LKS+13]. The reason is that, modern DRAM technology typically connects many storage cells with a single sense amplifier through a bitline, as shown in Figure 2.22(a). In this manner, many cells can be sensed with a relatively small number of sense amplifier, but at the cost of long bitlines. These long bitlines have a high parasitic capacitance, which is the dominant source of DRAM latency [LKS+13]. Therefore, specialized low-latency DRAMs typically use shorter bitlines at the cost of larger area overhead for sense amplifiers considering the same DRAM capacity, as shown in Figure 2.22(b). In summary, *DRAMs with longer bitlines have lower cost-per-bit and higher latency, while the ones with shorter bitlines have higher cost-per-bit and shorter latency.*

To achieve the benefits of both low access latency and low cost-per-bit, previous work has proposed *Tiered-Latency DRAM* (TL-DRAM), whose key idea is to split long bitline in each DRAM subarray into two shorter segments using an *isolation transistor* [LKS+13], as shown in Figure 2.22(c). Therefore, one segment (i.e., near segment) in each bitline of TL-DRAM can be accessed with the latency of a short-bitline, while incurring minimum area and cost-per-bit overheads.

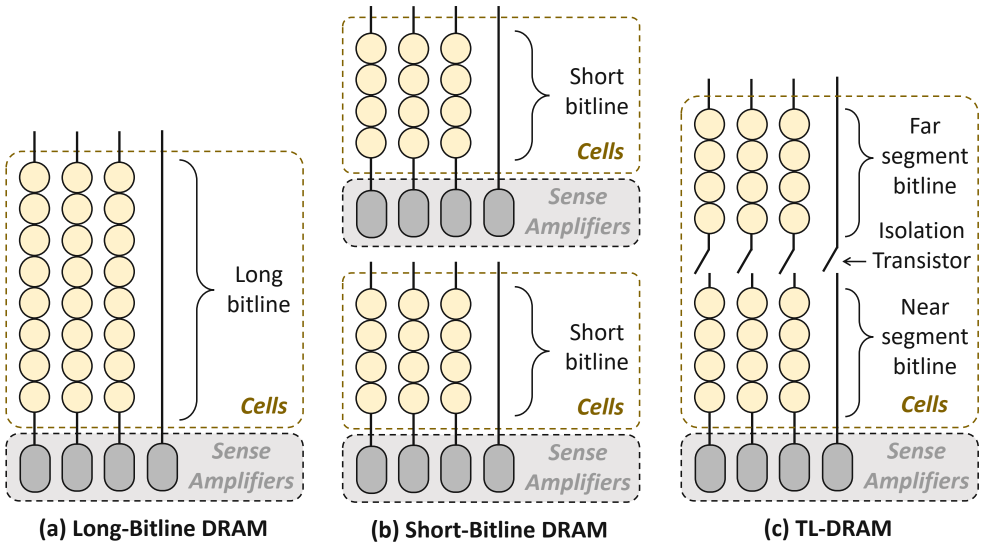


Figure 2.22: An overview of the architectural differences among (a) long-bitline DRAM, (b) short-bitline DRAM, and (c) TL-DRAM, considering the same number of cells-per-bitline (adapted from [LKS+13]).

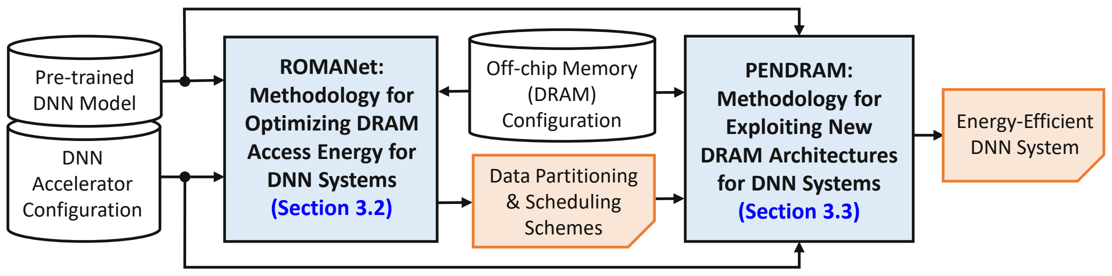
## 2.7 Summary of Background and Related Work

This chapter mainly discusses the fundamentals of NNs (i.e., DNNs and SNNs), reliability threats in computing systems, and DRAM technology. First, this chapter provides detailed description of DNNs and SNNs, encompassing their operations, models, training and inference process, and HW accelerators. Then, it presents state-of-the-art techniques for improving the energy efficiency of the DNN and SNN systems. This chapter also provides detailed description of reliability threats in computing systems including approximation errors, permanent faults, and soft errors. Then, it presents state-of-the-art techniques for mitigating the reliability threats in SNN systems. Afterward, this chapter describes the quantization techniques for NNs, and discusses the DRAM technology as the main memory of modern computing systems, including DRAM organization, operations, and novel architectures. Furthermore, this chapter also identifies the limitations of state-of-the-art techniques for improving energy efficiency and fault tolerance of NN-based systems, which will be addressed in the following chapters.

CHAPTER 3

# DRAM Optimization for Energy-Efficient DNN Systems

This chapter discusses our novel methodology to perform DRAM optimization for achieving energy-efficient DNN systems. This chapter first identifies and discusses the problems for enabling energy-efficient DNN inference systems in Section 3.1. To systematically address the research problems, we propose a novel methodology that employs our proposed HW/SW-level optimization techniques for achieving energy-efficient DNN systems. The proposed design flow is shown in Figure 3.1 and the details of novel contributions are described in the following sections in this chapter. Section 3.2 discusses a novel methodology for optimizing the DRAM access requirements for DNN accelerators. It employs a design space exploration (DSE) to find the appropriate data partitioning and scheduling scheme that minimizes the number of DRAM accesses for the given DNN model. Furthermore, Section 3.3 discusses a novel methodology for exploiting the characteristics of new DRAM architectures through a generalized DRAM data mapping for optimizing the DRAM energy-per-access for DNN accelerators, thereby improving the energy efficiency of DNN systems.



Pre-trained DNN Model
DNN Accelerator Configuration
ROMANet: Methodology for Optimizing DRAM Access Energy for DNN Systems (Section 3.2)
Off-chip Memory (DRAM) Configuration
Data Partitioning & Scheduling Schemes
PENDRAM: Methodology for Exploiting New DRAM Architectures for DNN Systems (Section 3.3)
Energy-Efficient DNN System

Figure 3.1: An overview of the design flow of this chapter.

## 3.1 Problem Identification

CNNs, a particular type of DNNs, have emerged as a promising solution for a wide range of machine learning applications, e.g., image classification, object detection, autonomous vehicles, and smart healthcare [LBH15]. To expedite the inference process, several CNN HW accelerators have been proposed over the past few years [CDS+14, ZLS+15, ZDZ+16, HLM+16, CKES17, AJH+16, LLL+16, JYP+17, PRM+17, LYL+17, MUDV17, YYY+18, KSK18, HPT+18, SPS+18, UAH+18, LKK+18, ZLL+21, SNFK20]. These accelerators offer a higher performance efficiency as compared to the general-purpose CPU-based solution. However, most of them only present *isolated* accelerator design, and do not thoroughly study the impact of off-chip memory accesses, especially for scenarios where the full CNN processing cannot be mapped to an accelerator fabric at the same time. The reason is that CNN models are large in size, and typical CNN compute engine and on-chip memory are small and may not even be sufficient to process one complete layer of a network at a time. Moreover, each data is usually involved in multiple computations (i.e., MAC operations) during processing. Therefore, multiple redundant accesses for the same data to off-chip memory are inevitable. The redundant accesses to DRAM-based off-chip memory hinder the accelerators from achieving further efficiency gains as DRAM energy is significantly higher compared to other operations [SCYE17, PHS20, CBM+20b, CBM+20a, AAH+20], which is also shown in Figure 2.7(b). Therefore, *DRAM energy savings result in the proportional system-level energy savings*. DRAM access energy is dependent on the number of accesses and the energy-per-access that varies depending upon whether the access faces a row buffer hit, a row buffer miss, or a row buffer conflict, as described in Section 2.6.2. We observe that, a row buffer hit incurs less latency and energy as compared to a row buffer miss and a row buffer conflict, as shown in Figure 3.2. From these observations, it is evident that *DRAM access energy an latency for CNN accelerators can be optimized by decreasing the number of DRAM accesses, row buffer conflicts and row buffer misses.*

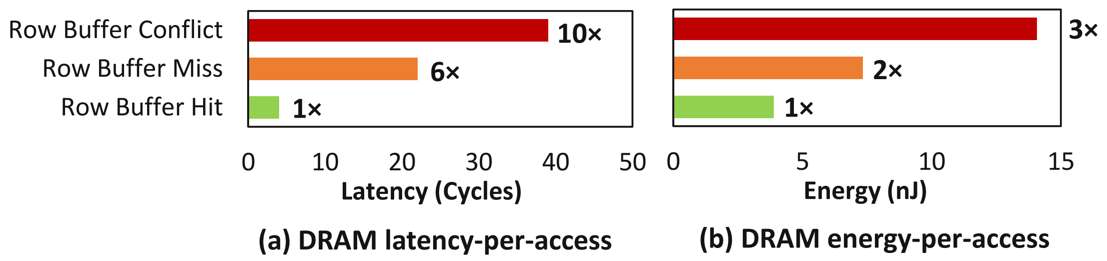


Figure 3.2: Profile of (a) DRAM access latency and (b) DRAM access energy for a row buffer conflict, miss, and hit. Data are obtained from our experiments for DDR3-1600 2Gb x8 using cycle-accurate DRAM simulators [KYM16, GYG+18].

The full CNN processing usually cannot be mapped at once to the accelerator fabric due to limited on-chip memory, i.e., 100 KB - 500 KB [SCYE17]. Therefore, to reduce the DRAM accesses, state-of-the-art works [ZLS+15, LYL+18] employ different *data* *partitioning and scheduling schemes* to move the data from DRAM to on-chip memory, and then reuse it multiple times for computation. Such data partitioning and scheduling schemes depend on which data type should be kept longer on chip, as there are three data types: input activations/feature maps (*ifmaps*), output activations/feature maps (*ofmaps*), and *weights*. Previous works can be loosely classified into two categories: *fixed scheduling* and *adaptive scheduling*. In fixed scheduling, the priority of data reuse is fixed to only one data type, which is not effective to minimize the DRAM accesses, since different layers of a network may have different priorities of data reuse. This limitation is addressed by the adaptive scheduling [LYL$^+$18, TKP20], the state-of-the-art scheduling which adaptively prioritizes the data reuse for each layer of a network. However, they do not consider the DRAM organization and overlapping data which should not be re-fetched again from DRAM in the analytical model for estimating the number of DRAM accesses. Therefore, *the analytical model used by state-of-the-art works for estimating the number of DRAM accesses is sub-optimal and needs re-formulation.*

Furthermore, CNN accelerators typically employ DRAM *burst mode* to increase the DRAM throughput [ZSF$^+$19, QWY$^+$16, GLX$^+$17], since it allows accessing multiple data with a single DRAM request. The state-of-the-art work [ZSF$^+$19] exploits the burst mode by mapping each data partition to continuous addresses in a DRAM bank. It prioritizes mapping each data partition to different columns in the same row of the same bank. If all columns in the same row are fully filled, the remaining data are mapped to a different row in the same bank. Therefore, each data partition may occupy multiple rows in a bank. However, this mapping has a high chance to face row buffer conflicts, since data from multiple rows in a bank need to be fetched for accessing the entire data partition, thereby consuming high energy and latency. Therefore, *the state-of-the-art works have not optimized the DRAM data mapping, which is crucial for CNN accelerators.*

**Problem-1:** *How can we optimize the number of DRAM accesses and the DRAM access energy for CNN accelerators, while improving the DRAM throughput.*

From the literature, there are various types of DRAM architectures that can be employed in CNN accelerators, including commodity DRAMs like DDR3 and DDR4 (see Section 2.6.3) and new DRAM architectures like SALP [KSL$^+$12] and TL-DRAM [LKS$^+$13] (see Section 2.6.4). In fact, different DRAM architectures have different characteristics regarding their access energy and access latency, thereby providing different energy efficiency gains for CNN accelerators. Moreover, new DRAM architectures (e.g., SALP and TL-DRAM) have been proposed to address some limitations of commodity DRAMs (e.g., latency) with some overheads (e.g., area, power/energy), thereby having the potential for improving the efficiency of CNN accelerators. Therefore, to maximize the energy efficiency gains of the given DRAM architecture in CNN accelerators, an understanding on the characteristics of the DRAM energy-per-access and latency-per-access is required. However, *state-of-the-art works have not studied these characteristics and potentials of different DRAM architectures for CNN accelerators, thereby limiting their energy efficiency gains.*

**Problem-2:** *How can we optimize the DRAM energy-per-access and latency-per-access of different DRAM architectures for maximizing the efficiency gains of CNN accelerators.*

**Benefits:** The solution to these problems will enable *energy-efficient DNN inference systems for energy-constrained embedded platforms and their applications for Edge-AI and Smart CPS*. *Edge-AI* is the system that runs AI algorithms on resource- and energy-constrained computing devices at the edge of the computing network, i.e., close to the source of data [CR19, SCZ+16, Sat17, YLH+18, LTL+19, CLMS20]. Meanwhile, *Smart CPS (Cyber-Physical System)* is the system that includes the interacting networks of computational components (e.g., computation and storage devices), physical components (e.g., sensors and actuators), and human users [GGWB17, CPS17, SKR18, KRH+18].

**Proposed Solution:** To systematically address the above problems, we propose a comprehensive solution which is discussed in several sections. Specifically, Problem-1 and Problem-2 are addressed in Section 3.2 and Section 3.3, respectively.

## 3.2 ROMANet: Reuse-Driven Off-Chip Memory Access Management for DNN Accelerators

This section aims at addressing **Problem-1**, i.e., the solution for optimizing the number of DRAM accesses and the DRAM access energy for CNN accelerators, while improving the DRAM throughput.

### 3.2.1 Motivational Study

If we consider the number of times a specific data is reused for MAC operations (i.e., *reuse factor*), then we will observe that different data types (i.e., *weights*, *ifmaps*, and *ofmaps*) may have different *reuse factors* across layers of a CNN model. Therefore, a data type that has the highest reuse factor in one layer can have the least reuse factor in another layer. For instance, our observations in Figure 3.3(a) illustrate that the 1st layer of VGG-16 (CONV1) has the highest reuse factor for the *weights*, however in the 14th layer of VGG-16 (FC1), *weights* has the lowest reuse factor. This indicates that, *the data partitioning and scheduling for DRAM accesses should consider the reuse factor information from each layer of a network, thereby maximizing data reuse on-chip and avoiding redundant off-chip DRAM accesses.* Furthermore, we also observe that the overlapping data on the *ifmaps* during the convolution can be stored on-chip and used in the subsequent process directly; see Figure 3.3(b). Therefore, these data do not need to be re-accessed again. In this manner, the redundant accesses to DRAM for the overlapping data can be minimized.

### 3.2.2 Scientific Research Challenges

An effective data partitioning and scheduling solution leads to the minimum number of DRAM accesses and thereby minimum energy consumption. However, the number of

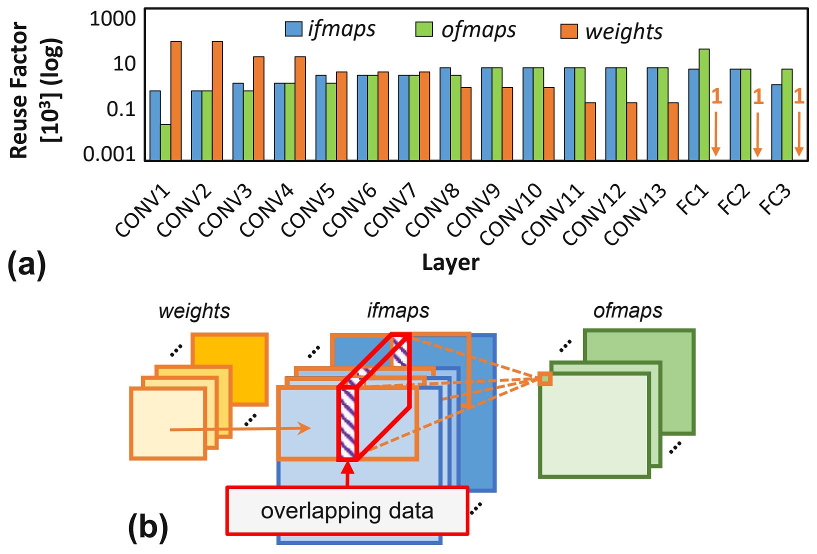
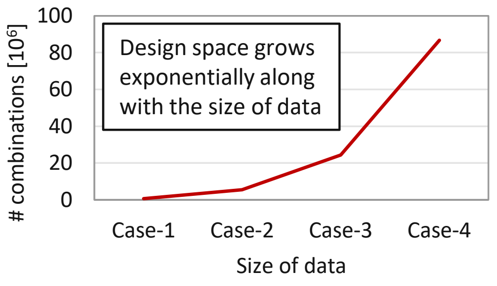


Figure 3.3: (a) Reuse factors for different data types in the VGG-16. (b) Illustration of overlapping data during convolutional processing.



| | Parameters of *ifmaps* and *weights* | | | | | |
|---|---|---|---|---|---|---|
| | ***H*** | ***W*** | ***I*** | ***P*** | ***Q*** | ***J*** |
| **Case-1** | 56 | 56 | 3 | 3 | 3 | 8 |
| **Case-2** | 112 | 112 | 3 | 3 | 3 | 16 |
| **Case-3** | 168 | 168 | 3 | 3 | 3 | 32 |
| **Case-4** | 224 | 224 | 3 | 3 | 3 | 64 |

# combinations = $H \times W \times I \times P \times Q \times J$

**Legend** *H*: height of *ifmaps*, *W*: width of *ifmaps*, *I*: depth of *ifmaps*/*weights*, *P*: height of *weights*, *Q*: width of *weights*, *J*: number of *weights*

Figure 3.4: Estimated number of data partitioning options to be investigated in the design space (left) for different cases in the table. Each option defines the size of *ifmaps* and *weights* partitions to be fetched from the DRAM and used for computing the *ofmaps*.

possible configurations can be significantly large and grows exponentially with the data size, as shown in Figure 3.4. Therefore, it is necessary to efficiently explore the design space in search of effective solution(s). Toward this, we need to formulate an optimization problem, i.e., we have to identify the constraints and formulate the design goal. Solving this problem requires an analytical model to efficiently estimate the number of DRAM accesses in an accurate manner.

***Required:*** *A methodology that optimizes the number of DRAM accesses through effective data partitioning and scheduling, and optimizes the DRAM energy-per-access through efficient DRAM data mapping.*

### 3.2.3 Novel Contributions

To address the above challenges, we propose *ROMANet, a novel methodology that enables fine-grained Reuse-driven Off-chip Memory Access management for deep neural Network accelerators* [PHS21b], that employs the following key techniques; see an overview in Figure 3.5.

1. **An analytical model** to compute the number of DRAM accesses for a given data partitioning and scheduling scheme of a network layer. It models (1) the reuse factors of different data types (*ifmaps*, *ofmaps*, and *weights*), and (2) the data partitions.
2. **A data mapping in off-chip DRAM** that maximizes the row buffer hits, bank- and chip-level parallelism, and exploits DRAM multi-bank burst feature. This mechanism considers data partitioning and DRAM configuration as inputs.
3. **A data mapping in on-chip SRAM buffers** that exploits bank-level parallelism to efficiently shuttle data between DRAM and compute engine. This mechanism considers data partitioning, buffer, and compute engine configuration as inputs.

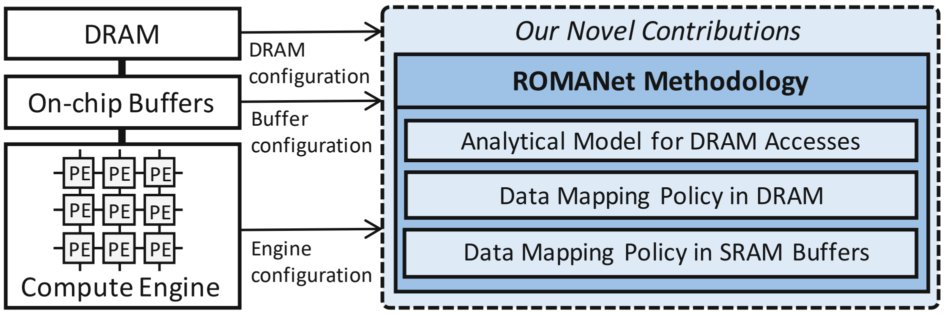


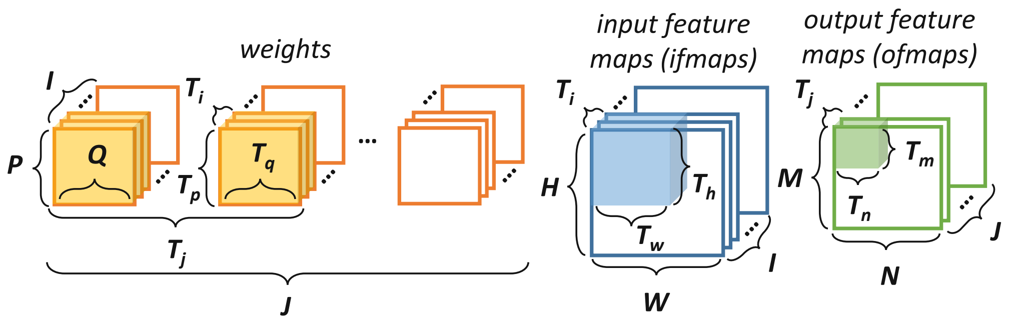
Figure 3.5: Overview of our novel contributions, as highlighted in blue boxes.

### 3.2.4 Data Partitioning and Scheduling

The data partitioning and scheduling of convolutional operations in a CNN are typically devised for each layer of a network, hence most of the CNN accelerators process a network sequentially, one layer at a time [ZLS+15, CKES17, LLL+16]. The pseudo-code of a convolutional layer processing in a CNN accelerator can be represented as shown in Figure 3.6(a). It has two main parts: the inner loops and the outer loops. The inner loops represent the processing of a portion of the convolutional layer whose data is available in the on-chip memory. The outer loops define the schedule of processing different portions. The size of a portion that can be processed at one time depends on the data that can be stored in the on-chip memory. Therefore, based on the size of the on-chip memory, the data can be partitioned in the form of blocks/tiles as tiling can improve locality when there is data reuse [WL91], and this is represented with the step sizes in the outer loops in Figure 3.6(a). The sequence of the outer loops defines the sequence in which these tiles are moved from the DRAM to the on-chip memory to process the corresponding

```
for ( z = 0; z < Z; z++ ) {  // Z: number of images in a batch      Off-chip data transfer
  for ( m = 0; m < M; m += T_m ) {  // M: height of ofmaps              (outer loops)
    for ( n = 0; n < N; n += T_n ) {  // N: width of ofmaps
      for ( j = 0; j < J; j += T_j ) {  // J: depth of ofmaps
        for ( i = 0; i < I; I += T_i ) {  // I: depth of ifmaps & weights
          // load ifmaps, weights, and ofmaps (psums)
          for ( p = 0; p < P; p++ ) {  // P: height of weights           On-chip computation
            for ( q = 0; q < Q; q++ ) {  // Q: width of weights             (inner loops)
              for ( mx = m; mx < min(mx+T_m, M); mx++ ) {
                for ( nx = n; nx < min(nx+T_n, N); nx++ ) {
                  for ( jx = j; jx < min(jx+T_j, J); jx++ ) {
                    for ( ix = i; ix < min(ix+T_i, I); ix++ ) {
                      ofmaps[z][mx][nx][jx] += …
                      ifmaps[z][str*mx+p][str*nx+q][ix] * weights[p][q][ix][jx]
          } } } } } }
          // store ofmaps (psums)
} } } } }
```

**(a) Pseudo-code of a tiled convolutional layer processing**



| | | |
|---|---|---|
| $T_p$ : tile height of *weights* | $T_h$ : tile height of *ifmaps* | $T_m$ : tile height of *ofmaps* |
| $T_q$ : tile width of *weights* | $T_w$ : tile width of *ifmaps* | $T_n$ : tile width of *ofmaps* |
| $T_i$ : tile depth of *weights* & *ifmaps* | | $T_j$ : tile depth of *ofmaps* |

**(b) Illustration of a tiled convolutional layer processing**

Figure 3.6: (a) Pseudo-code of a tile-based convolutional layer processing. (b) Illustration of a tile-based convolutional layer processing. Tiling factors define the portion of data (shaded region) in each data type which are used for on-chip processing at one time.

portions of the layer. This sequence also defines the total number of DRAM accesses required to process a layer. Here, two subsequent on-chip computations can have some data in common that do not need re-fetching from the DRAM, and thereby affecting the total number of DRAM accesses required for inference.

To minimize DRAM accesses, different data partitioning and scheduling have been studied and employed by state-of-the-art CNN accelerators. The idea is to keep data (that has to be reused the most) longer in the on-chip memory. Depending upon which data type should be kept longer in the on-chip memory, the previous works can be loosely classified into two main categories: *fixed scheduling* and *adaptive scheduling*.

**Fixed scheduling** employs static tiling factors and scheduling across layers of a network, by giving priority of reuse to only one specific data type, either *ifmaps*, *ofmaps*, or *weights*. This concept has been widely used in previous works, such as [ZLS+15]. However, the reuse factors of different data types vary across layers of a CNN model. Therefore, a data type that has the highest reuse factor in one layer can have the least reuse factor in another layer. This can lead to a significant loss in energy efficiency, specifically when a fixed dataflow forces a data type which has the highest reuse factor, to be fetched multiple times from DRAM, instead of keeping it longer in the on-chip memory and reusing it to the maximum level.

To address this limitation, **adaptive scheduling** is proposed in the literature [LYL+18, TKP20]. They employ adaptive tiling factors and scheduling, thereby further reducing the DRAM accesses. However, their analytical models for estimating the number of DRAM accesses do not consider the DRAM organization and the overlapping data which should not be re-fetched from DRAM. Therefore, their analytical models for estimating the number of DRAM accesses are sub-optimal and need to be re-formulated. Furthermore, they also do not consider the DRAM data organization to reduce the DRAM energy-per-access and latency-per-access.

### 3.2.5 ROMANet Methodology

We propose the ROMANet methodology [PHS21b] to provide a novel synergistic optimization to improve the DRAM access energy and data throughput in CNN accelerators; see Figure 3.7. The overview of its operational flow is explained in the following points.

- First, we **determine the data partitioning and scheduling scheme** for each layer of a network that offers the minimum DRAM accesses while considering numerous parameters, which is a non-trivial problem. To do this, a design space exploration (DSE) is performed. Here, the DSE employs the analytical model of DRAM accesses while considering different sizes of data partitioning for all data types and scheduling schemes, as well as leveraging the information of the CNN model (e.g., data size and stride), the DRAM (e.g., number of banks), and the on-chip accelerator fabric (e.g., buffer size). Furthermore, we devise an analytical model that considers minimizing the redundant accesses for overlapping data.
- We **devise en efficient data mapping policy in DRAM** that places each data partition across the available DRAM channels, ranks, chips, banks, rows, and columns, in a manner to maximize the row buffer hits, bank- and chip-level parallelism, while exploiting the DRAM multi-bank burst feature.
- We also **devise en efficient data mapping policy in SRAM buffers** that places each data partition across the available SRAM banks, rows, and columns, in a manner to maximize the bank-level parallelism.

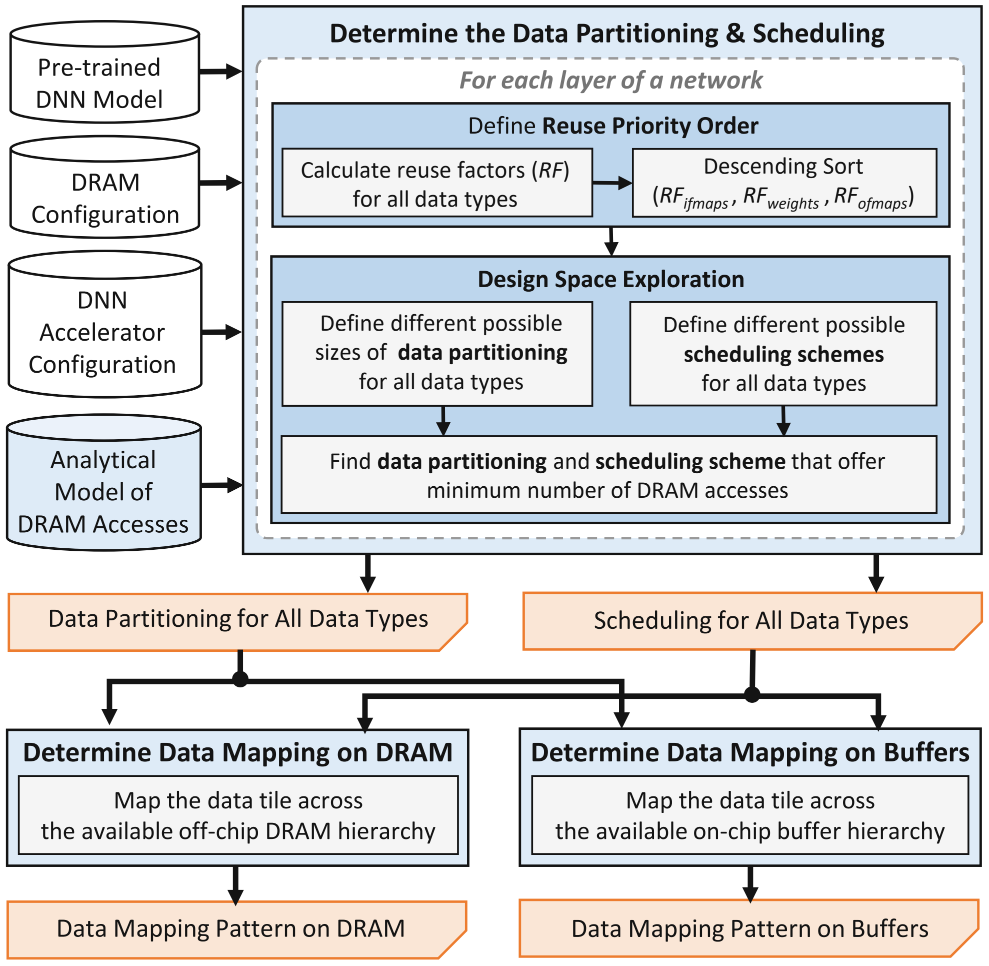

Pre-trained DNN Model
DRAM Configuration
DNN Accelerator Configuration
Analytical Model of DRAM Accesses
Determine the Data Partitioning & Scheduling
For each layer of a network
Define Reuse Priority Order
Calculate reuse factors (RF) for all data types
Descending Sort ($RF_{ifmaps}$, $RF_{weights}$, $RF_{ofmaps}$)
Design Space Exploration
Define different possible sizes of data partitioning for all data types
Define different possible scheduling schemes for all data types
Find data partitioning and scheduling scheme that offer minimum number of DRAM accesses
Data Partitioning for All Data Types
Scheduling for All Data Types
Determine Data Mapping on DRAM
Map the data tile across the available off-chip DRAM hierarchy
Determine Data Mapping on Buffers
Map the data tile across the available on-chip buffer hierarchy
Data Mapping Pattern on DRAM
Data Mapping Pattern on Buffers

Figure 3.7: An overview of the ROMANet methodology. The novel contributions are highlighted in blue.

#### 3.2.5.1 Analytical Modeling for Estimating the Number of DRAM Accesses

In the ROMANet, there are two key solutions proposed: (1) DSE, and (2) memory mapping in DRAM and SRAM buffers. To perform DSE, we develop analytical models for (a) reuse factors, (b) data partitioning, and (c) DRAM accesses. These models jointly provide a measure of the number of DRAM accesses that should be minimized. Figure 3.8 provides an overview of how these models are connected.

**Design Space Reduction:** We reduce the overall design space to be investigated by fixing some design parameters.

- We consider a CNN accelerator design based on Tensor Processing Unit (TPU) [JYP+17], with single-level SRAM buffers hierarchy implemented using *scratch-pad memory* (SPM), as shown in Figure 2.5. We consider SPM since it is commonly used as the local buffer in many CNN accelerators [SCB19].

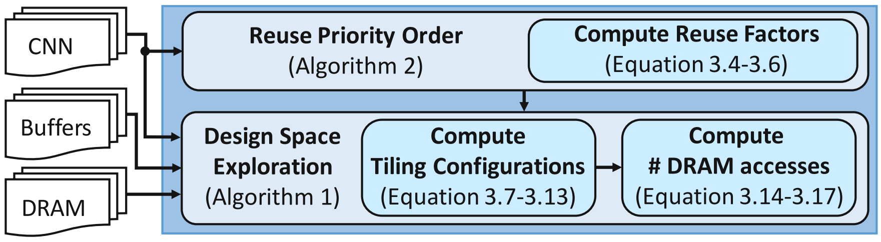

CNN
Buffers
DRAM
Reuse Priority Order (Algorithm 2)
Compute Reuse Factors (Equation 3.4-3.6)
Design Space Exploration (Algorithm 1)
Compute Tiling Configurations (Equation 3.7-3.13)
Compute # DRAM accesses (Equation 3.14-3.17)

Figure 3.8: Flow of steps to compute the number of DRAM accesses.

- We process only a single layer of a network at one time.
- We also consider to process a tile of *ifmaps* and *weights* on-chip, and producing a tile of *ofmaps* at one time. A tile of *ifmaps* is defined by *tiling factors* $T_h \cdot T_w \cdot T_i$, a tile of *weights* is defined by $T_p \cdot T_q \cdot T_i \cdot T_j$, and a tile of *ofmaps* is defined by $T_m \cdot T_n \cdot T_j$, as shown in Figure 3.6(b).

**The Optimization Problem Formulation:** We formulate the optimization problem to minimize the total number of DRAM accesses for a network ($\#DRAM_{access}$). The $\#DRAM_{access}$ is defined as the sum of the DRAM accesses from all layers of the network, expressed as

$$\#DRAM_{access} = \sum_{l=1}^{L} \#DRAM^{l}_{access} \tag{3.1}$$

The $\#DRAM^{l}_{access}$ represents the total number of DRAM accesses in layer-$l$, and $L$ represents the number of layers in the network. *Since we consider to process only a single layer of a network at one time, the optimization of $\#DRAM_{access}$ can be approached by minimizing the number of DRAM accesses-per-layer ($\#DRAM^{l}_{access}$).* We consider the data partitioning approach in the optimization problem, as it is employed by many CNN accelerators, such as [ZLS+15, CKES17, LYL+17]. Therefore, the objective of the optimization is formulated as

$$\begin{aligned}
&Objective: \underset{<T_h,T_w,T_i,T_j,T_m,T_n>}{minimize} \#DRAM^{l}_{access}\\
&Constraints:\\
&\qquad (T_h \cdot T_w \cdot T_i) \cdot bit_{ifmaps} \le iBuff\\
&\qquad (T_p \cdot T_q \cdot T_i \cdot T_j) \cdot bit_{weights} \le wBuff\\
&\qquad (T_m \cdot T_n \cdot T_j) \cdot bit_{ofmaps} \le oBuff
\end{aligned} \tag{3.2}$$

Constraints of the optimization problem are the size of on-chip buffers, because they limit the volume of data that can be stored at one time. *iBuff*, *wBuff*, and *oBuff* denote the sizes of *ifmaps* buffer, *weight* buffer, and *ofmaps* buffer, respectively. Here, the $\#DRAM^{l}_{access}$ is defined as a sum of the number of DRAM accesses from all data types in layer-$l$, and can be expressed as

$$\#DRAM^{l}_{access} = \#access^{l}_{ifmaps} + \#access^{l}_{weights} + \#access^{l}_{ofmaps} \tag{3.3}$$

Terms $\#access^l_{ifmaps}$, $\#access^l_{weights}$, and $\#access^l_{ofmaps}$ represent the number of DRAM accesses in layer-$l$ for *ifmaps*, *weights*, and *ofmaps*, respectively. From the above equations, it is evident that to further define the model in a more fine-grained manner, the formulations need to consider the modeling of different data types for calculating the reuse factors, the data partitioning, and the DRAM accesses.

### 1) Model of Reuse Factors

The reuse factor ($RF$) represents the number of MACs that each data entity is used for [CYES19]. Hence, each data type has its own reuse factor in each layer of a network, which can be estimated by

$$RF^l_{ifmaps} = \left\lceil \frac{P^l}{str} \right\rceil \cdot \left\lceil \frac{Q^l}{str} \right\rceil \cdot J^l \tag{3.4}$$

$$RF^l_{weights} = \left\lceil \frac{H^l - P^l + 1}{str} \right\rceil \cdot \left\lceil \frac{W^l - Q^l + 1}{str} \right\rceil \tag{3.5}$$

$$RF^l_{ofmaps} = P^l \cdot Q^l \cdot I^l \tag{3.6}$$

Terms $P^l$ and $Q^l$ denote the height and the width of *weights*; $I^l$ and $J^l$ denote the number of *ifmaps* and *ofmaps*; $H^l$ and $W^l$ denote the height and the width of *ifmaps*; while $str$ represents stride, respectively. Each of which is for a specific layer-$l$. The *reuse factor* computations for different data types are illustrated in Figure 3.9.

### 2) Model of Data Partitioning

#### 2.a) Input Feature Maps (*ifmaps*)

Depending upon the size of tiling factors, *ifmaps* may have many possibilities of data partitioning. Therefore, multiple tiles may have different sizes and there may be overlapping data that do not require redundant fetches. Here, we observe that there are three possible access directions that can affect the modeling of layer partitioning: *width-wise*, *height-wise*, and *depth-wise*, as shown in Figure 3.10.

Width-wise direction: In this case, tile access scheduling for width-wise direction is prioritized. For instance, if we start the access from tile-1 of *ifmaps*, then the sequence of access is tile-1 $\rightarrow$ tile-2 $\rightarrow$ tile-3. In this direction, there may be three types of tiles involved (i.e., tile-1, tile-2, and tile-3), which differ from each other in the tiling width ($T_w^{'}$: base tiling width, $T_{w_{int}}$: intermediate tiling width, and $T_{w_{last}}$: last tiling width). The model is formulated as

$$T_{w_{last}} = W - T_w^{'} - n_{T_{w_{int}}} \left( T_w^{'} - T_{w_{ov}} \right) \tag{3.7}$$

Term $n_{T_{w_{int}}}$ denotes the number of intermediate tile ($T_{w_{int}}$). The $T_{w_{int}}$ value reflects a portion of the overlapped data ($T_{w_{ov}}$) from previous fetch that should not be re-fetched again, because $T_{w_{int}} = T_w^{'} - T_{w_{ov}}$.

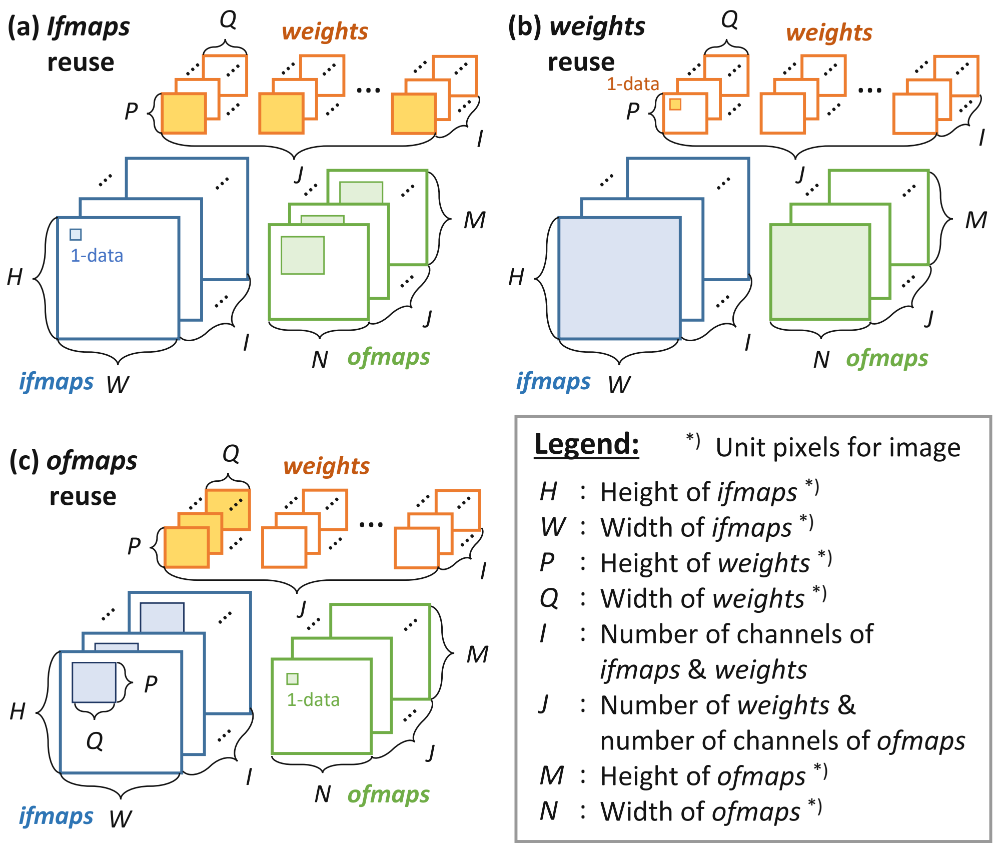
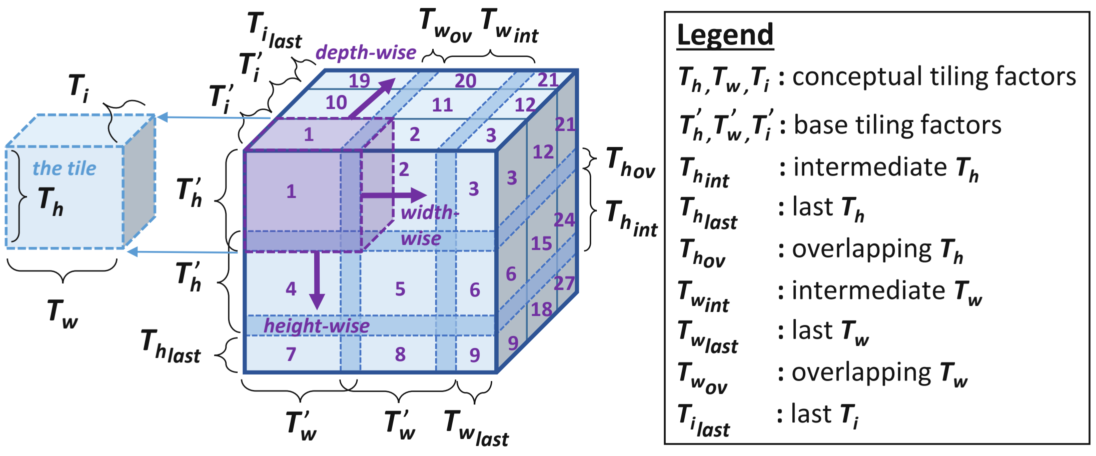

(a) *Ifmaps* reuse
weights
Q
P
I
J
M
H
1-data
ifmaps
W
I
N
ofmaps
J
(b) *weights* reuse
1-data
(c) *ofmaps* reuse
Legend: *) Unit pixels for image
H : Height of *ifmaps* *)
W : Width of *ifmaps* *)
P : Height of *weights* *)
Q : Width of *weights* *)
I : Number of channels of *ifmaps* & *weights*
J : Number of *weights* & number of channels of *ofmaps*
M : Height of *ofmaps* *)
N : Width of *ofmaps* *)

Figure 3.9: The reuse factor calculations in different data types involve the corresponding shaded regions. (a) In $RF_{ifmaps}$: A single pixel of *ifmaps* is reused as many as the number of pixels within the shaded region in *weights*. (b) In $RF_{weights}$: A single *weight* is reused as many as the number of pixels within the shaded region in *ifmaps*. (c) In $RF_{ofmaps}$: A single pixel of *ofmaps* is reused as many as the number of *partial sums* produced from convolution between the shaded regions in *ifmaps* and *weights*.

<u>Height-wise direction:</u> In this case, tile access scheduling for height-wise direction is prioritized. For instance, if we start the access from tile-1 of *ifmaps*, then the sequence of access is tile-1 → tile-4 → tile-7. In this direction, there may be three types of tiles involved (i.e., tile-1, tile-4, and tile-7), which differ from each other in the tiling height ($T_h^{'}$: base tiling height, $T_{h_{int}}$: intermediate tiling height, and $T_{h_{last}}$: last tiling height). The model is formulated as

$$T_{h_{last}} = H - T_h^{'} - n_{T_{h_{int}}}\left(T_h^{'} - T_{h_{ov}}\right) \tag{3.8}$$

Term $n_{T_{h_{int}}}$ denotes the number of intermediate tile ($T_{h_{int}}$). The $T_{h_{int}}$ value reflects a portion of the overlapped data ($T_{h_{ov}}$) from previous fetch that should not be re-fetched again, because $T_{h_{int}} = T_h^{'} - T_{h_{ov}}$.



$T_{i_{last}}$
$T_{w_{ov}}$ $T_{w_{int}}$
depth-wise
$T_i'$
$T_i$
the tile
$T_h$
$T_w$
$T_h'$
$T_{h_{last}}$
width-wise
height-wise
$T_w'$ $T_w'$ $T_{w_{last}}$
$T_{h_{ov}}$
$T_{h_{int}}$
1 2 3 4 5 6 7 8 9 10 11 12 15 18 19 20 21 24 27
Legend
$T_h, T_w, T_i$ : conceptual tiling factors
$T_h', T_w', T_i'$ : base tiling factors
$T_{h_{int}}$ : intermediate $T_h$
$T_{h_{last}}$ : last $T_h$
$T_{h_{ov}}$ : overlapping $T_h$
$T_{w_{int}}$ : intermediate $T_w$
$T_{w_{last}}$ : last $T_w$
$T_{w_{ov}}$ : overlapping $T_w$
$T_{i_{last}}$ : last $T_i$

Figure 3.10: Model of layer partitioning for *ifmaps*.

Depth-wise direction: In this case, tile access scheduling for depth-wise direction is prioritized. For instance, if we start the access from tile-1 of *ifmaps*, then the sequence of access is tile-1 → tile-10 → tile-19. In this direction, there may be two possible types of tiles involved (i.e., tile-1 and tile-19), which differ from each other in the tiling depth ($T_i^{'}$: base tiling depth and $T_{i_{last}}$: last tiling depth). Here, there is no intermediate tile because of no overlapping data. The model is formulated as

$$T_{i_{last}} = I - n_{T_i} \cdot T_i^{'} \tag{3.9}$$

Term $n_{T_i}$ denotes the number of tile $T_i^{'}$. In summary, the tiling configuration of whole *ifmaps* is characterized by a combination of tiling factors $T_h \in \{T_h^{'}, T_{h_{int}}, T_{h_{last}}\}$, $T_w \in \{T_w^{'}, T_{w_{int}}, T_{w_{last}}\}$, and $T_i \in \{T_i^{'}, T_{i_{last}}\}$.

### 2.b) Filter Weights (*weights*)

For *weights*, the data partitioning is characterized by $T_p$, $T_q$, $T_i$, and $T_j$ as shown in Figure 3.11. Since the height and the width of *weights* are typically small, we simplify the $T_p$ and $T_q$ as $T_p = P$ and $T_q = Q$ to reduce the complexity of the model and design space. Therefore, there are only two possible access directions: *filter set-wise direction* and *depth-wise direction*.

Filter set-wise direction: In this case, tile access scheduling for filter set-wise direction is prioritized. The sequence of access is tile-1 → tile-2 → tile-3. In this direction, there may be two possible types of tiles involved, which differ from each other in the number of filters in a filter set ($T_j^{'}$: base tiling set and $T_{j_{last}}$: last tiling set). Here, there is no intermediate tile because of no overlapping data. The model is formulated as

$$T_{j_{last}} = J - n_{T_j} \cdot T_j^{'} \tag{3.10}$$

Term $n_{T_j}$ denotes the number of filter set $T_j^{'}$.

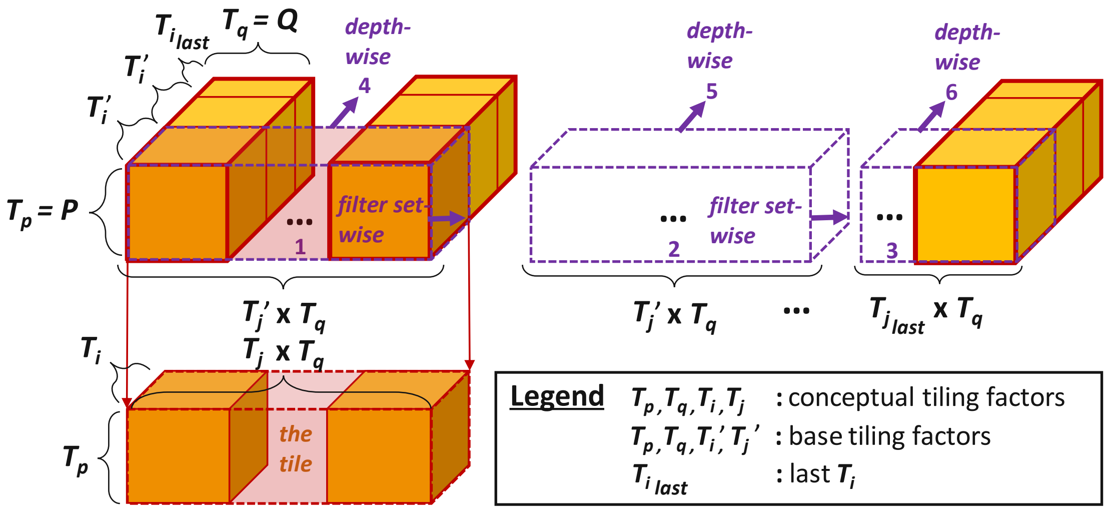


Figure 3.11: Model of layer partitioning for *weights*.

Depth-wise direction: In this case, tile access scheduling for depth-wise direction is prioritized. The sequence of access is tile-1 → tile-4. In this direction, there may be two possible types of tiles involved, which differ from each other in the tiling depth ($T_i^{'}$: base tiling depth and $T_{i_{last}}$: last tiling depth). Here, there is no intermediate tile because of no overlapping data. The model is formulated as

$$T_{i_{last}} = I - n_{T_i} \cdot T_i^{'} \tag{3.11}$$

Term $n_{T_i}$ denotes the number of tile depth $T_i^{'}$. In summary, tiling configuration of *weights* is characterized by a combination of tiling factors $T_p \in \{P\}$, $T_q \in \{Q\}$, $T_i \in \{T_i^{'}, T_{i_{last}}\}$, and $T_j \in \{T_j^{'}, T_{j_{last}}\}$.

### 2.c) Output Feature Maps (*ofmaps*)

For *ofmaps*, the data partitioning is characterized by $T_m$, $T_n$, and $T_j$, as illustrated in Figure 3.12. A tile of *ofmaps* is generated from MAC operations between a tile of *ifmaps* and a tile of *weights* at one time. Which tile of *ofmaps* generated from MACs, depends on which tile of *ifmaps* and *weights* that are computed. Hence, the model of *ofmaps* in terms of tiling height $T_m$ and tiling width $T_n$, can be formulated as

$$T_m = \left\lceil \frac{T_h - T_p + 1}{str} \right\rceil \tag{3.12}$$

$$T_n = \left\lceil \frac{T_w - T_q + 1}{str} \right\rceil \tag{3.13}$$

Meanwhile, the model for tiling depth $T_j$ in *ofmaps* follows the Equation 3.10. Even though the *ifmaps* may have intermediate values such as intermediate height $T_{h_{int}}$ and intermediate width $T_{w_{int}}$, its MAC computations will still consider base tiling height $T_h^{'}$ and base tiling width $T_w^{'}$ due to overlapping data. Therefore, in *ofmaps*, the generated $T_m$ is limited to $\{T_m^{'}, T_{m_{last}}\}$ and $T_n$ is limited to $\{T_n^{'}, T_{n_{last}}\}$. In summary, the layer partitioning of the *ofmaps* is defined by combination of tiling factors $T_m \in \{T_m^{'}, T_{m_{last}}\}$, $T_n \in \{T_n^{'}, T_{n_{last}}\}$, and $T_j \in \{T_j^{'}, T_{j_{last}}\}$.

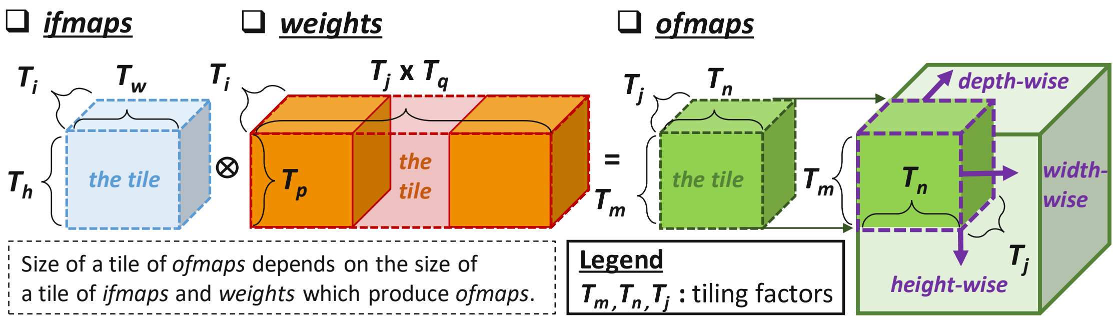


Figure 3.12: Model of layer partitioning for *ofmaps*.

## 3) Model of DRAM Accesses

Equation 3.3 shows that the DRAM access-per-layer is defined as the sum of the DRAM accesses from all data types. Meanwhile, the number of DRAM accesses for each data type is dependent on the data partitioning. Therefore, the number of DRAM accesses-per-layer for each data type can be estimated as the sum of the number of DRAM accesses-per-tile, and can be formulated as

$$\#access_x^l = \sum_{t=1}^{N_{T_x}} \#access_x^t \qquad \text{with} \;\; x \in \{ifmaps, weights, ofmaps\} \tag{3.14}$$

Term $\#access_x^t$ represents the number of DRAM accesses-per-tile for $x$ data type ($x$ is either *ifmaps*, *weights*, or *ofmaps*), while $N_{T_x}$ denotes the number of tiles in layer-$l$, for $x$ data type. In DRAM, *ifmaps* and *weights* have only read ($rd$) type of accesses, thus their number of accesses-per-tile can be estimated as

$$\#access_{ifmaps}^t = \left\lceil \frac{T_h \cdot T_w \cdot T_i}{D_p} \right\rceil_{rd} \tag{3.15}$$

$$\#access_{weights}^t = \left\lceil \frac{T_p \cdot T_q \cdot T_i \cdot T_j}{D_p} \right\rceil_{rd} \tag{3.16}$$

Term $D_p$ denotes the number of DRAM chips-per-rank. Meanwhile, the *ofmaps* may have two types of DRAM accesses, read ($rd$) and write ($wr$). These two types exist when a tile of *partial sums* which are in *oBuff* can not be accumulated with newly generated *partial sums*, but they still need to be accumulated with other *partial sums* to produce final *ofmaps*. Hence, they need to be stored back to DRAM, so that the *oBuff* can provide space to store a new tile of *partial sums*. Later, this tile of *partial sums* has to be fetched from DRAM to complete the computation, producing a tile of final *ofmaps*. The generic equation to estimate the number of DRAM accesses-per-tile for *ofmaps* is formulated as

$$\#access^{t}_{ofmaps} = \left\lceil \frac{T_m \cdot T_n \cdot T_j}{D_p} \right\rceil_{rd} + \left\lceil \frac{T_m \cdot T_n \cdot T_j}{D_p} \right\rceil_{wr} \tag{3.17}$$

Equation 3.17 shows that the same *partial sums* are moved from *oBuff* to DRAM ($wr$) and fetched again from DRAM to *oBuff* ($rd$). If there is no need for transporting *partial sums* back to DRAM, the equation only needs to consider the write access part ($wr$) for storing final *ofmaps* to DRAM.

#### 3.2.5.2 Proposed Design Space Exploration (DSE)

*We devise an algorithm that performs an exhaustive DSE for searching the effective data partitioning and scheduling that offer minimum DRAM accesses.* The algorithm is presented in Algorithm 1 and explained in the following steps.

**Step-1:** For each layer of a network, we define scheduling schemes to be explored in the DSE, which are based on the reuse priority orders that are determined using Algorithm 2. Here, the reuse factors of different data types are calculated and then sorted, so that the order of priority is known (Algorithm 2: lines 3-5). The idea is to ensure that the data type with a higher reuse factor has a higher priority to be kept longer in the on-chip buffer and reused maximally for reducing the redundant accesses to DRAM. The possible orders are presented in Table 3.1. Therefore, a network will have reuse priority orders from all layers which can be used to define scheduling schemes for DSE (Algorithm 1: line 3). For instance, if a layer has reuse priority order of *ofmaps* → *ifmaps* → *weights*, then the corresponding scheduling is devised as shown in Figure 3.13. Here, the priority is to maximally reuse *ofmaps* or partial sums (*psums*) by traversing the tile accesses of *ifmaps* and *weights* in the depth-wise direction. Therefore, the DRAM accesses for *ofmaps* are minimized, i.e., only for storing the final *ofmaps* to the DRAM. Furthermore, *ifmaps* should have a less number of redundant accesses than *weights*, since *ifmaps* has higher reuse priority than *weights*. It is expected to be achieved by putting the tiling parameters of *ifmaps* (i.e., $nT_h$, $nT_w$) at the outer loop of the scheduling.

Table 3.1: Possible reuse priority orders for scheduling.

| **Reuse Factors** | | | | | |
|---|---|---|---|---|---|
| **Highest** | **Medium** | **Lowest** | **Highest** | **Medium** | **Lowest** |
| *ifmaps* | *weights* | *ofmaps* | *weights* | *ofmaps* | *ifmaps* |
| *ifmaps* | *ofmaps* | *weights* | *ofmaps* | *ifmaps* | *weights* |
| *weights* | *ifmaps* | *ofmaps* | *ofmaps* | *weights* | *ifmaps* |

**Step-2:** Fetch a tile of *ifmaps* and *weights* from DRAM, as long as they fit in the corresponding buffer (*iBuff* and *wBuff*, respectively) and can be used together in MAC

**Algorithm 1** Pseudo-code of the proposed DSE

**INPUT: (1)** CNN: #layers ($L$), *ifmaps* ($H$, $W$, $I$), *weights* ($P$, $Q$, $I$), *ofmaps* ($M$, $N$, $J$), etc.;
**(2)** Buffer size: *ifmaps* (*iBuff*), *weights* (*wBuff*), *ofmaps* (*oBuff*);
**(3)** Bitwidth: *ifmaps* ($bit_{ifm}$), *weights* ($bit_{wgh}$), *ofmaps* ($bit_{ofm}$);
**(4)** Analytical models: (i) tiling factors of *ifmaps* ($T_h$, $T_w$, $T_i$), *weights* ($T_p$, $T_q$, $T_i$, $T_j$), *ofmaps* ($T_m$, $T_n$, $T_j$); (ii) DRAM accesses;
**(5)** Reuse Priority Orders (*RPO*);
**OUTPUT: (1)** Number of DRAM accesses ($\#DR_{access}$);
**(2)** Data partitioning: *ifmaps* ($TP_{ifm}$), *weights* ($TP_{wgh}$), *ofmaps* ($TP_{ofm}$);
**(3)** Scheduling (*Schedule*);

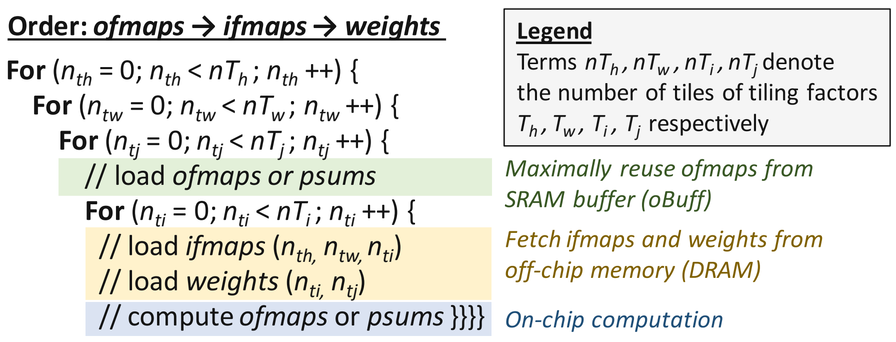
```
BEGIN
    Initialization:
 1: $T_p = P$;
 2: $T_q = Q$;
    Process:
 3: $\#Scheduling \leftarrow RPO$;
 4: for ($Layer = 1$ to $L$) do
 5:   for ($Sched = 1$ to $\#Scheduling$) do
 6:     for ($T_h = P : step_{T_h} : H$) do
 7:       for ($T_w = Q : step_{T_w} : W$) do
 8:         for ($T_j = 1 : step_{T_j} : J$) do
 9:           Calculate $T_i$;
10:           if ($T_h \cdot T_w \cdot T_i \cdot bit_{ifm} \leq iBuff$) and ($P \cdot Q \cdot T_i \cdot T_j \cdot bit_{wgh} \leq wBuff$) then
11:             Calculate $T_m$, $T_n$, $T_j$;
12:             if ($T_m \cdot T_n \cdot T_j \cdot bit_{ofm} \leq oBuff$) then
13:               Calculate $\#DR_{access}$;
14:               if (first loop) then
15:                 $minDR_{access} = \#DR_{access}$;
16:                 Save $TP_{ifm}$, $TP_{wgh}$, and $TP_{ofm}$;
17:                 Save $Schedule = Sched$;
18:               else if ($\#DR_{access} \leq minDR_{access}$) then
19:                 $minDR_{access} = \#DR_{access}$;
20:                 Save $TP_{ifm}$, $TP_{wgh}$, and $TP_{ofm}$;
21:                 Save $Schedule = Sched$;
22: return (1) $minDR_{access}$, (2) $TP_{ifm}$, $TP_{wgh}$, $TP_{ofm}$, (3) $Schedule$;
END
```

operations for generating a tile of *ofmaps*, whose size has to fit in the *oBuff* (Algorithm 1: lines 10-12). To define the size of a tile, we explore different combinations of tiling factors from *ifmaps*, *weights*, and *ofmaps* (Algorithm 1: lines 6-12). Here, *we consider the adjustable search steps for different tiling factors, i.e., $step_y$, with $y \in \{T_h, T_w, T_j\}$, to achieve faster search, as the higher search step means smaller design space to be explored.* This gives trade-offs between the possible number of DRAM accesses that can be found and the DSE computation time.

**Step-3:** DSE calculates the number of DRAM accesses (in Algorithm 1: line 13) based on the layer partitioning and scheduling, which have been defined in the previous steps.

**Algorithm 2** Pseudo-code of the Reuse Priority Order

```
INPUT: (1) CNN: #layers ($L$);
    (2) Analytical model of reuse factor: ifmaps ($RF_{ifm}$), weights ($RF_{wgh}$), ofmaps ($RF_{ofm}$);
OUTPUT: Reuse Priority Orders ($RPO$); // for Algorithm 1
BEGIN:
    Initialization:
 1: $RF_{ifm}, RF_{wgh}, RF_{ofm} = 0$;
 2: $RPO = []$;
    Process:
 3: for ($Layer = 1$ to $L$) do
 4:     Calculate $RF_{ifm}$, $RF_{wgh}$, $RF_{ofm}$;
 5:     $RPO[Layer] \leftarrow Sort(RF_{ifm}, RF_{wgh}, RF_{ofm})$;
 6: return $RPO$;
END:
```



Order: ofmaps → ifmaps → weights
For ($n_{th}$ = 0; $n_{th}$ < $nT_h$ ; $n_{th}$ ++) {
For ($n_{tw}$ = 0; $n_{tw}$ < $nT_w$ ; $n_{tw}$ ++) {
For ($n_{tj}$ = 0; $n_{tj}$ < $nT_j$ ; $n_{tj}$ ++) {
// load ofmaps or psums
For ($n_{ti}$ = 0; $n_{ti}$ < $nT_i$ ; $n_{ti}$ ++) {
// load ifmaps ($n_{th}$, $n_{tw}$, $n_{ti}$)
// load weights ($n_{ti}$, $n_{tj}$)
// compute ofmaps or psums }}}}
Legend
Terms $nT_h$, $nT_w$, $nT_i$, $nT_j$ denote the number of tiles of tiling factors $T_h$, $T_w$, $T_i$, $T_j$ respectively
Maximally reuse ofmaps from SRAM buffer (oBuff)
Fetch ifmaps and weights from off-chip memory (DRAM)
On-chip computation

Figure 3.13: Illustration of a scheduling scheme derived from the reuse priority order of *ofmaps* → *ifmaps* → *weights*. Here, *ofmaps* is maximally reused by re-fetching the *partial sums* from *oBuff* to produce final *ofmaps*.

**Step-4:** The information of data partitioning and scheduling that offer minimum DRAM accesses is saved (Algorithm 1: lines 14-21), and then used for mapping data to DRAM and buffers, which are discussed in Sections 3.2.5.3 and 3.2.5.4, respectively.

**Note**: The DSE needs to be performed only once at the design time to find the effective partitioning and scheduling policy. Once the policy has been found, the corresponding settings in the CNN accelerator are set once during the initialization stage before performing an inference.

### 3.2.5.3 Data Mapping in the DRAM

As the dataflow of CNN processing is known prior to the execution, it is always certain at every point during the execution which data will be required next. Therefore, CNN execution can significantly benefit from the spatial locality. Spatial locality means that if a particular data is referenced at a particular time, it is likely that the adjacent data will be referenced in the near future. In CNN execution where the dataflow is completely known, *the spatial locality can be maximized by mapping the data that will be required in the subsequent cycles together in the DRAM.*

To understand the energy and latency values associated with a single DRAM access, we perform experiments to observe the energy and latency incurred by a single DRAM access in different conditions, i.e., ❶ a row buffer miss, ❷ a row buffer hit, ❸ a row buffer conflict, and ❹ access to a location in a different bank in the same chip (see Figure 3.14). Here, we consider DDR3-1600 2Gb x8 DRAM (i.e., 1 channel, 1 rank-per-channel, 1 chip-per-rank, 8 banks-per-chip, and 8-bit I/O). We employ the state-of-the-art DRAM simulator [KYM16] for simulating the DRAM, and employ the DRAM energy simulator [GYG+18] for estimating the DRAM access energy. The energy and latency results are shown in Figure 3.14(b) and Figure 3.14(c), respectively.

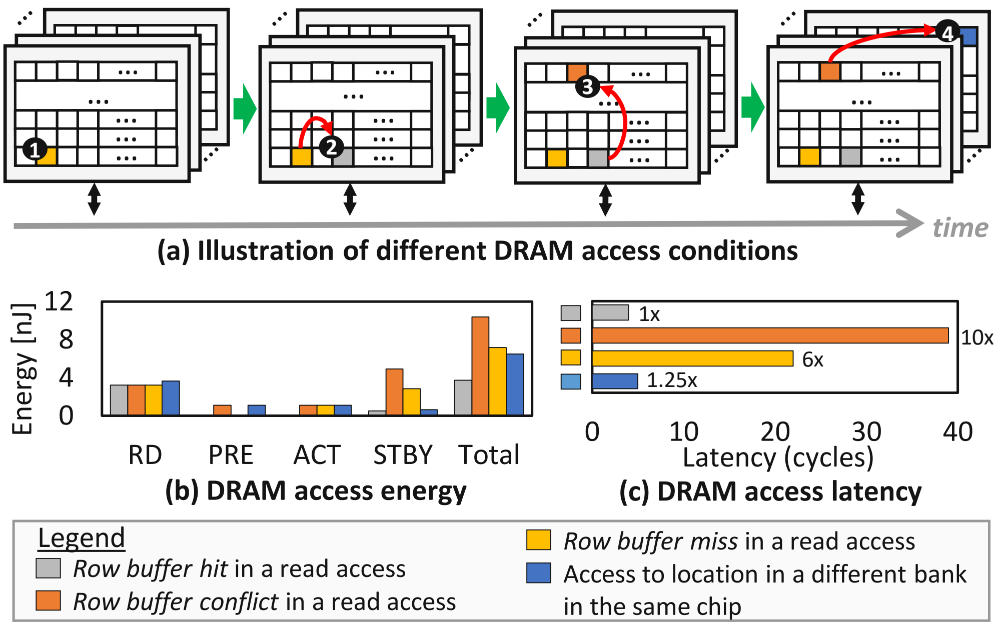


Figure 3.14: (a) Illustration of different DRAM access conditions and their corresponding (b) DRAM access energy and (c) DRAM access latency.

These figures show some key observations:

- A single DRAM access consumes *standby* (STBY) energy and operational energy: *read* (RD) or *write* (WR), *activation* (ACT), and *precharge* (PRE) [GLH+19].
- A row buffer hit requires *read/write* and *standby* energy, thereby consuming less access energy and latency as compared to a row buffer conflict or a row buffer miss.
- A row buffer conflict requires *precharge*, *activation*, and *standby* energy, as it has to close the currently activated row and then open a different row. It consumes the highest access energy and latency among the observed cases.
- A row buffer miss requires *activation* and *standby* energy to activate the target row as there is no activated row yet.

- Exploiting DRAM bank-level parallelism (i.e., access to location in a different bank of the same chip in a short time interval) can also be done faster and consume less energy as compared to a row buffer conflict. It requires the *activation*, *precharge* and *standby* energy with less total energy than a row buffer conflict due to its lower latency.

Furthermore, DRAM has the multi-bank burst feature that can be exploited to further improve the data throughput, as shown in Figure 3.15. Hence, we also leverage the DRAM multi-bank burst feature to devise an effective DRAM mapping that offers high throughput with the lowest access energy and latency.

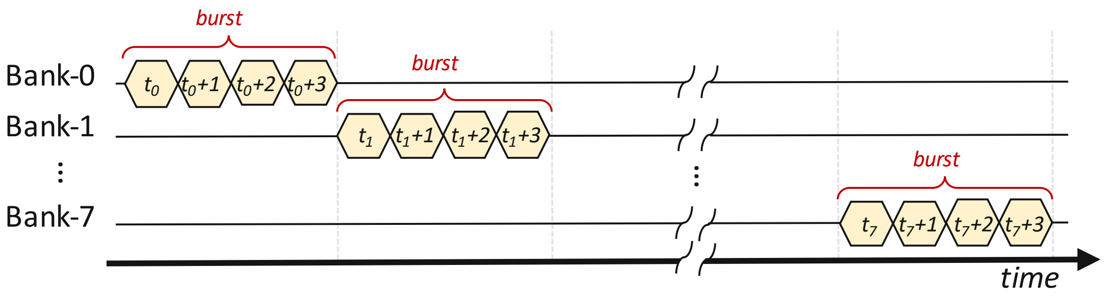
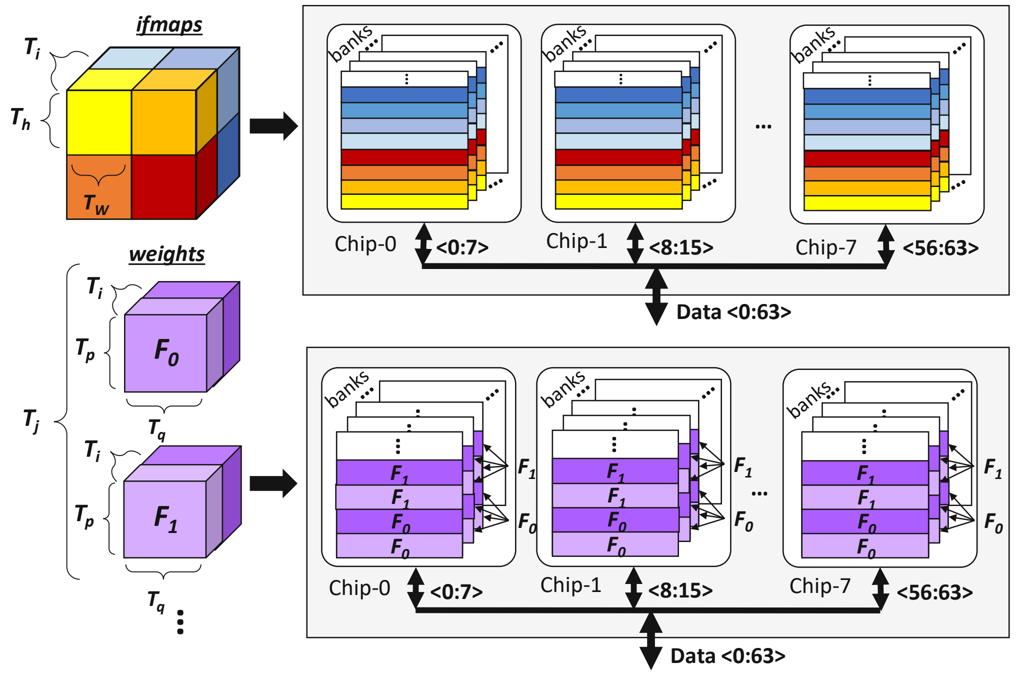


Figure 3.15: Timing diagram of the DRAM multi-bank burst feature, which happens in the same DRAM chip.

**Proposed DRAM Data Mapping Policy:** Here, multiple data from the same tile which are expected to be accessed subsequently in a short time interval, are placed in the same row of the same bank, thereby increasing the row buffer hits. To further minimize the row buffer conflicts while increasing the throughput, chip- and bank-level parallelism are exploited. Exploiting chip-level parallelism means that if multiple data from the same tile are expected to be fetched in parallel, these data are placed across chips, if applicable. Similarly, for bank-level parallelism, the data are placed across banks in the same chip, if applicable. Our DRAM mapping also exploits the DRAM multi-bank burst feature to further improve the DRAM throughput. This mapping concept is illustrated in Figure 3.16(a).

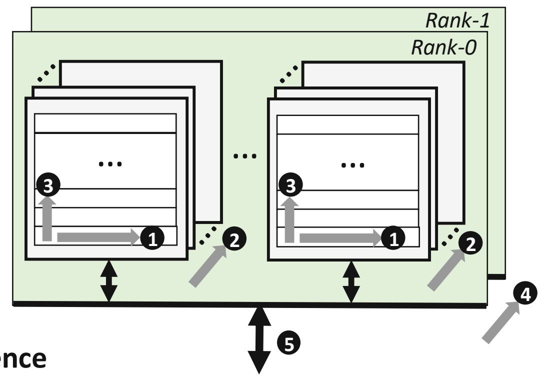
We further define the DRAM mapping sequence, based on our observations in Figure 3.14. The proposed mapping sequence is shown in Figure 3.16(b). In step-❶, we prioritize mapping a data tile to different columns in the same row, to achieve maximum row buffer hits. This step can be done across different chips in parallel, if applicable, to exploit chip-level parallelism. If all columns in the same row are fully filled, then the remaining data are mapped to the different banks in the same chip, to exploit bank-level parallelism (step-❷). Mapping to different banks can also be done across different chips in parallel, if applicable. For each bank, data are mapped to different columns in the same row, just like step-❶. Here, if all columns in the same row across all banks are fully filled, then the remaining data are mapped to a different row (step-❸). These steps ❶-❸ are repeated until all data are mapped in a DRAM rank. If there are remaining data left, they can be mapped to different ranks (step-❹) and channels (step-❺) respectively if applicable, using the same steps as ❶-❸.



**(a) Overview of DRAM mapping for *ifmaps* and *weights* across banks and chips**

Pseudo-code of DRAM mapping

```
❺ For (ch = 0,  ch < #channel , ch++) {
 ❹ For (ra = 0,  ra < #rank , ra++) {
  ❸ For (ro = 0,  ro < #row , ro++) {
   ❷ For (ba = 0, ba < #bank , ba++) {
    ❶ For (co = 0, co < #column , co++) {
       // map a tile of data to
          DRAM [ch, ra, ba, ro, co];
} } } } }
```



**(b) Overview of DRAM mapping sequence**

Figure 3.16: (a) Overview of the proposed DRAM data mapping. (b) Proposed DRAM data mapping sequence. In the pseudo-code, #column denotes the number of columns-per-row, #row denotes the number of rows-per-bank, #bank denotes the number of banks-per-chip, #rank denotes the number of ranks-per-module, and #channel denotes the number of channel.

To effectively perform DRAM mapping, information regarding data partitioning and scheduling for different data types is required. Note that our DRAM mapping scheme can be employed for all DRAM variants (e.g., DDR3, DDR4, etc.) since all of them have similar internal organization, i.e., they consist of channels, ranks, chips, banks, rows, and columns, when viewed from a top-down perspective (see Figure 2.19) [GLH$^+$19].

### 3.2.5.4 Data Mapping in the SRAM Buffer

To efficiently transport the data between the DRAM and the compute engine, a data mapping in the SRAM buffer is required. Since the DRAM mapping considers tile-wise mapping, hence the SRAM buffer considers the same. Here, we employ SRAM buffer in the scratch-pad memory fashion, since it is commonly used in many CNN accelerators as the local on-chip buffer. The idea of the mapping is that, if multiple data from the same tile are expected to be fetched in parallel, these data are placed across banks to exploit bank-level parallelism. Based on the SRAM buffer configuration (e.g., capacity and number of banks), we can devise an efficient data mapping.

**Proposed SRAM Data Mapping Policy:** The concept of the proposed mapping is illustrated in Figure 3.17. Here, we prioritize mapping a data tile (for each data type) to the same row, across different banks, to achieve maximum bank-level parallelism. If the same rows across different banks are fully filled, then the remaining data are mapped in a different row, across different banks. These steps are repeated until all data from the same tile are mapped in the SRAM buffer. In this manner, each tile may occupy multiple subsequent rows. For *weights* data type, if a systolic array-based CNN accelerator is considered, different filters can be mapped in different banks, thus each bank can supply specific filter(s) to a specific column of the systolic array. Furthermore, we can also employ different sectors in the buffer, and each having a dedicated sleep transistor to power-gate the unused sectors, for obtaining lower power and energy.

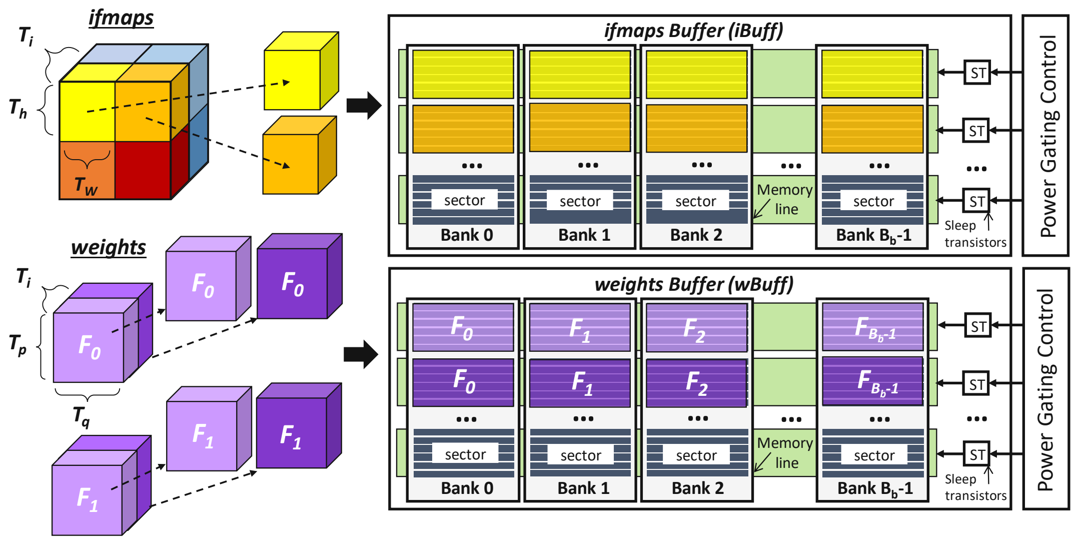


Figure 3.17: Overview of the concept in SRAM buffer data mapping for *ifmaps* and *weights*, across different banks.

### 3.2.6 Evaluation Methodology

Figure 3.18 shows our experimental setup and tool flow for evaluating our ROMANet methodology.

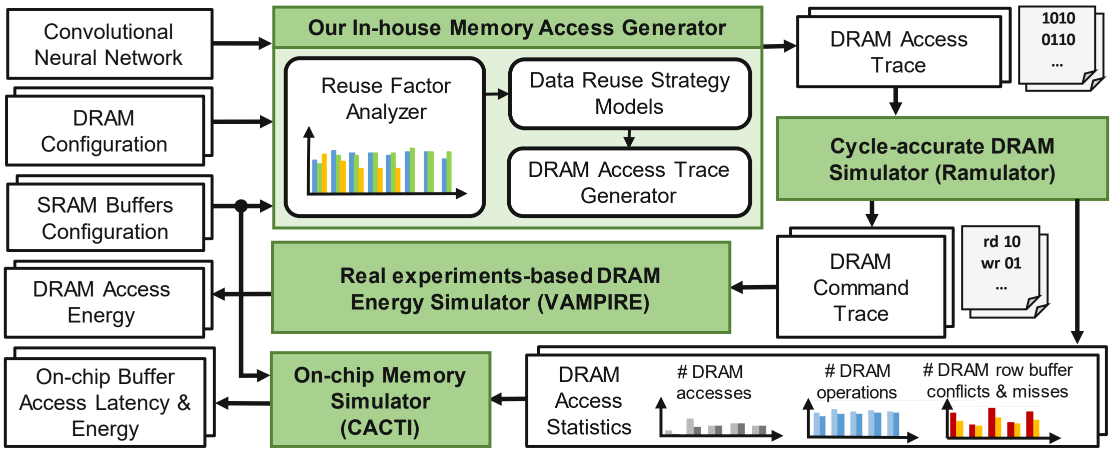

Convolutional Neural Network
DRAM Configuration
SRAM Buffers Configuration
DRAM Access Energy
On-chip Buffer Access Latency & Energy
Our In-house Memory Access Generator
Reuse Factor Analyzer
Data Reuse Strategy Models
DRAM Access Trace Generator
DRAM Access Trace
1010
0110
...
Cycle-accurate DRAM Simulator (Ramulator)
Real experiments-based DRAM Energy Simulator (VAMPIRE)
DRAM Command Trace
rd 10
wr 01
...
On-chip Memory Simulator (CACTI)
DRAM Access Statistics
# DRAM accesses
# DRAM operations
# DRAM row buffer conflicts & misses

Figure 3.18: Our experimental setup and tool flow.

**Memory Access Generator:** Our in-house memory access generator receives the CNN information (e.g., number of layers) and the configuration of DRAM and SRAM buffers, as the inputs. It produces a DRAM access trace that represents the sequence of DRAM requests. Our tool analyzes the characteristics of CNN and extracts its reuse factor information. The reuse factor is used for devising the scheduling, depending on the methodology. Afterward, the tool generates the representative DRAM access trace.

**DRAM Simulator:** We use the state-of-the-art and cycle-accurate DRAM simulator (i.e., Ramulator [KYM16]) to simulate the DRAM. The input is the DRAM access trace from our memory access generator. The outputs are DRAM command trace and statistics, such as the number of accesses, operations, row buffer conflicts and row buffer misses.

**DRAM Power/Energy Simulator:** To estimate the DRAM access energy for the defined DRAM requests, we use the state-of-the-art real experiments-based DRAM energy simulator (i.e., VAMPIRE [GYG+18]). The input is the DRAM command trace generated by the DRAM simulator and the output is the DRAM access energy estimation.

**On-chip Memory Simulator:** To estimate the on-chip buffers' access latency and energy, we used the state-of-the-art on-chip memory simulator (i.e., CACTI 7.0 [BKM+17]) that incorporates the number of on-chip buffer accesses and the on-chip buffer latency-per-access and energy-per-access.

In this evaluation, we consider a state-of-the-art TPU [JYP+17]-like CNN accelerator, as specified in Figure 2.5 and Table 3.2. We consider DRAM with DDR3-1600 2Gb x8 configuration [Mic10, MLN+12]. The DRAM *open row* policy was used as it keeps the row open after an access, hence the subsequent accesses to the same row can be performed fast and with less energy. We use first-come first-serve (FCFS) scheduling policy for handling DRAM requests, which serves the request that is received first. We use different DRAM access modes: (1) *burst mode* with burst length = 8 (BL8) for DDR3 DRAM [Mic10], which provides multiple words of data per-request, and (2) *non-burst mode*, which provides a single word of data per-request. For the networks, we use the AlexNet [KSH12] and the

VGG-16 [SZ14] to represent conventional CNNs, and the MobileNet [HZC+17] and the SqueezeNet [IHM+16] to represent CNNs for embedded applications. The SqueezeNet has fire modules, which are slightly different from the conventional CNNs. The conventional ones are typically composed of cascaded convolutional (CONV) layers, and each having uniform-sized filters. Meanwhile, each fire module consists of *a squeeze layer* and *an expand layer*. The squeeze layer is simply a CONV layer, while the expand layer is a CONV layer that contains 1x1 and 3x3 filters. The expand layer can be implemented as two separate CONV layers. Hence, the dimensions of these CONV layers can be inserted into the analytical model in ROMANet for the optimization process. ROMANet can also be used for residual networks (e.g., ResNet [HZRS16]), as shown in Figure 3.19. In each residual module, ROMANet considers the dimensions of the CONV layers for the optimization process. Here, *ifmaps* of the residual module are kept in the DRAM and accessed when they are used for CONV execution and element-wise addition. For the evaluation, we focus on the conventional CNNs and the ones for embedded applications.

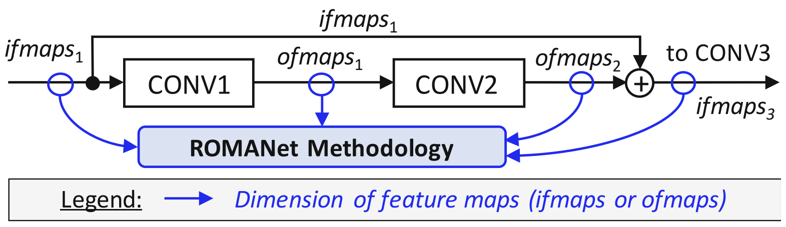


Figure 3.19: A single residual module in a residual network and the parts that benefit from our ROMANet.

Table 3.2: Configuration of the systolic array-based CNN accelerator.

| Module | Description |
|---|---|
| Systolic Array | Size = 8 × 8 PEs (1 PE = 1 MAC operation) |
| Buffers | 64 KB *iBuff*, 64 KB *wBuff*, 64 KB *oBuff*<br>#banks = 8 banks-per-buffer |
| Memory Controller | Policy = open row, Scheduler = FCFS |
| DRAM | DDR3-1600 2Gb x8<br>1 channel, 1 rank/channel, 1 chip/rank, 8 banks/chip<br>Mode: non-burst, burst (burst length = 8) |

To study the improvements offered by the ROMANet over the state-of-the-art works, i.e., Caffeine [ZSF+19], SmartShuttle [LYL+18], and Bus-Width Aware (BWA) approach [TKP20], we implement and integrate them in our experiments. For data reuse strategy, the Caffeine employs *ofmaps*-reuse scheduling, while the SmartShuttle and the BWA use *weights*-reuse and *ofmaps*-reuse scheduling. For DRAM mapping, since the SmartShuttle has no defined DRAM mapping scheme, we use the state-of-the-art mapping from the Caffeine for the SmartShuttle, which maps each layer partition in a continuous address space in a DRAM bank. The BWA employs similar DRAM mapping to effectively use the DRAM bandwidth.

### 3.2.7 Results and Discussion

This section discusses the experimental results regarding reductions in the number of DRAM accesses, DRAM requests, DRAM row buffer conflicts and misses, DRAM operations in Sections 3.2.7.1-3.2.7.4, subsequently. It then discusses the DRAM energy savings for dense and sparse CNNs in Sections 3.2.7.5-3.2.7.6, subsequently. It also discusses DRAM throughput and related observations in Sections 3.2.7.7-3.2.7.8, subsequently.

#### 3.2.7.1 Reduction in the Number of DRAM Accesses

Evaluation results for the number of DRAM accesses are presented in Figure 3.20. The results show that our ROMANet decreases the number of DRAM accesses over the BWA by 12% for the AlexNet, 36% for the VGG-16, 44% for the SqueezeNet, and 45% for the MobileNet. These improvements are achieved due to the effective data partitioning and scheduling performed by the ROMANet, and can be associated with several aspects.

1. The analytical model in the ROMANet considers the overlapping data that do not need to be re-fetched from the DRAM, while the analytical model of the BWA does not consider the overlapping data. It contributes 5% DRAM access reduction for the AlexNet, 16% for the VGG-16, 20% for the SqueezeNet, and 21% for the MobileNet.

2. The ROMANet also considers more possible scheduling schemes than the BWA, as they are devised from reuse factors and reuse priority orders. However, the BWA considers only the *weights*- and *ofmaps*-based scheduling schemes. It contributes 7% DRAM access reduction for the AlexNet, 20% for the VGG-16, 24% for the SqueezeNet, and 24% for the MobileNet.

Therefore, the ROMANet considers a wider search space than the state-of-the-art, as it investigates the number of DRAM accesses using a more detailed analytical model and more data partitioning and scheduling schemes, which in-turn open a higher possibility to find the configuration that leads to less number of DRAM accesses. These results also show that the reductions of the DRAM accesses happen on a layer-wise basis, which is in line with the defined optimization problem.

#### 3.2.7.2 Reduction in the Number of DRAM Requests

In practice, CNN accelerators typically exploit DRAM burst mode [ZSF+19, QWY+16, GLX+17]. Therefore, we also study the impact of the burst mode on the number of DRAM requests, compared to the non-burst mode. Evaluation results are provided in Figure 3.21. The results show that, the burst mode requires less number of DRAM requests as compared to the non-burst mode, for accessing the same number of data from DRAM. For instance, employing the burst mode with BL8 can decrease the number of DRAM requests by 8x. This is important because the burst mode only requires one-time address decoding to fetch multiple data, while the non-burst mode needs to perform address decoding each time a request is issued. Therefore, the burst mode incurs less access energy than the non-burst mode, which will be discussed in Section 3.2.7.5.

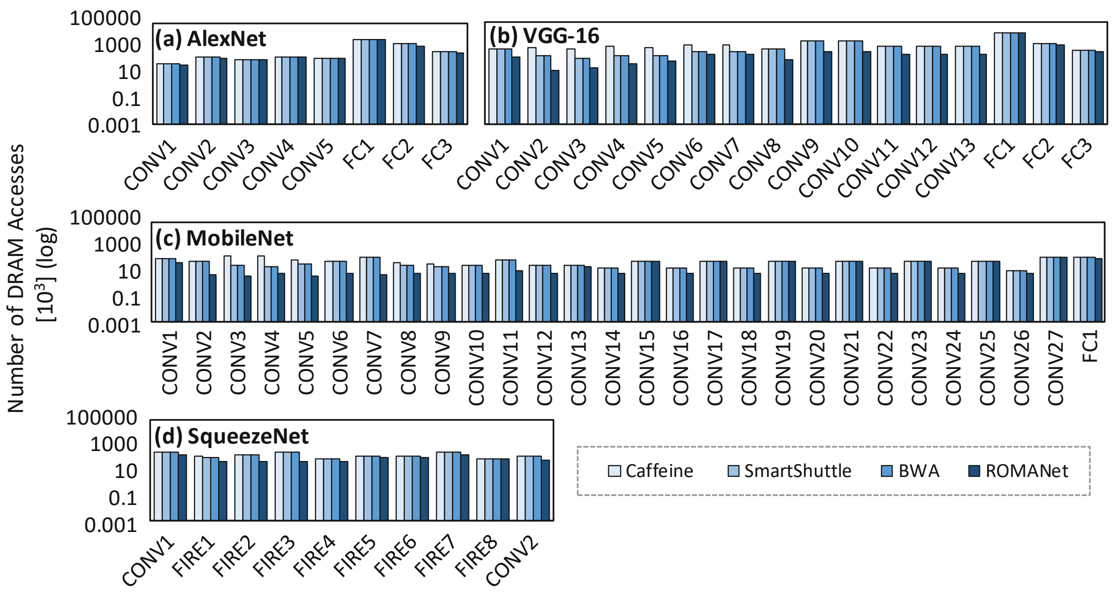


Figure 3.20: Results of the number of DRAM accesses for (a) AlexNet, (b) VGG-16, (c) SqueezeNet, and (d) MobileNet. Here, FIRE refers to the fire module.

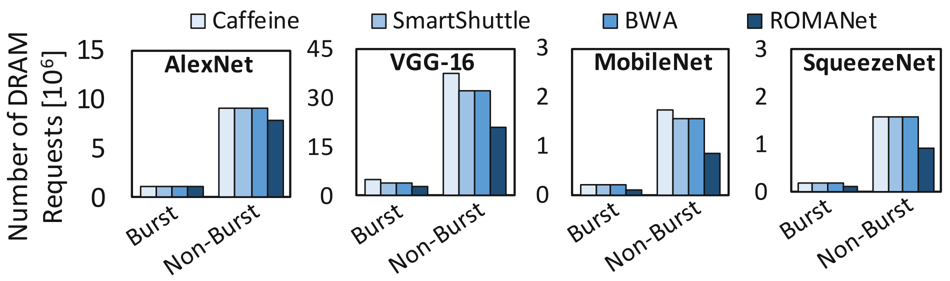


Figure 3.21: The total number of DRAM requests during an inference in different DRAM access modes (burst mode and non-burst mode).

### 3.2.7.3 Reduction in the Number of DRAM Row Buffer Conflicts and Misses

The evaluation results for the number of DRAM row buffer conflicts and misses are presented in Figure 3.22. The figure shows that, in general, our ROMANet decreases the number of DRAM row buffer conflicts and misses as compared to the BWA. For both burst mode and non-burst mode, ROMANet reduces the row buffer conflicts and misses by 12% for the AlexNet, 35% for the VGG-16, 48% for the MobileNet, and 45% for the SqueezeNet. These reductions are achieved because ROMANet devises an effective DRAM mapping policy that minimizes subsequent accesses to different rows in the same DRAM bank, when accessing a single tile partition. On the other hand, the addressing in the BWA makes the subsequent accesses to different rows in the same DRAM bank happen more frequent, when accessing a single tile partition. Therefore, it has a higher possibility of row buffer conflicts than the proposed technique.

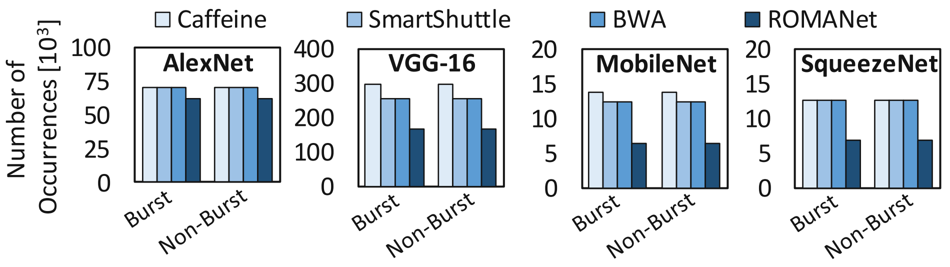


Figure 3.22: Results of the total number of DRAM row buffer conflicts and misses for the AlexNet, the VGG-16, the MobileNet, and the SqueezeNet.

### 3.2.7.4 Reduction in the Number of DRAM Operations

Reduction in the number of DRAM requests decreases the total number of DRAM operations, especially for *read* and *write*. Meanwhile, reduction in the number of DRAM row buffer conflicts and misses also decreases the total number of DRAM operations, especially for *activation* and *precharge*. The reason is that, if there is no row buffer conflict or miss, then there is no need for precharging and/or activating a different row in a DRAM bank. Moreover, the open row policy keeps the currently activated row open for some time to facilitate subsequent accesses to the same row at a faster rate and with less energy consumption. Therefore, the *precharge* and/or *activation* operations are not performed and the number of DRAM operations is reduced. It is validated by the evaluation results presented in Figure 3.23 for the AlexNet, the VGG-16, the MobileNet, and the SqueezeNet. Each bar of the graph in these figures already includes all DRAM operations: *activation*, *precharge*, *read*, and *write* operations.

Figure 3.23 shows that employing different DRAM access modes may lead to different numbers of operations. For both burst mode and non-burst mode, ROMANet reduces the operations by about 12% for the AlexNet, by about 34% for the VGG-16, by about 45% for the MobileNet, and by about 42% for the SqueezeNet. The burst mode has less number of DRAM operations as compared to the non-burst mode, since it has less number of DRAM requests (as shown in Figure 3.21). The reduction of DRAM operations is important because each operation consumes some energy. Therefore, reducing the number of DRAM operations leads to DRAM access energy savings, which will be discussed in Section 3.2.7.5.

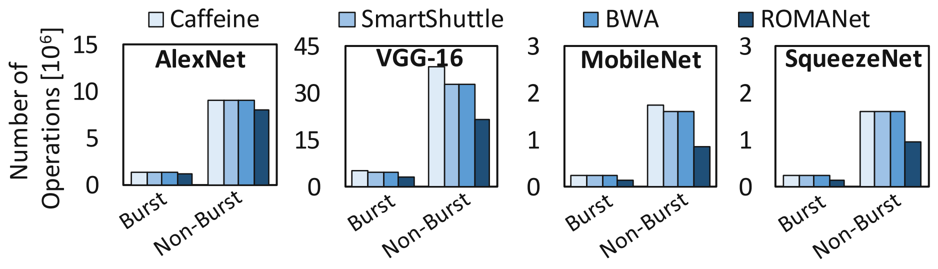


Figure 3.23: Results of the total number of DRAM operations performed for the AlexNet, the VGG-16, the MobileNet, and the SqueezeNet.

#### 3.2.7.5 DRAM Access Energy Savings

The evaluation results for the DRAM access energy are presented in Figure 3.24. The figure shows that, ROMANet reduces the DRAM access energy significantly as compared to the BWA, i.e., by 12% for the AlexNet, 36% for the VGG-16, 46% for the MobileNet, and 45% for the SqueezeNet. The burst mode has less DRAM access energy as compared to the non-burst mode (about 8x energy saving), since it has less number of DRAM operations. The main sources of these savings are: (1) the reduction in the number of accesses, and (2) the reduction in the number of row buffer conflicts and misses. The energy savings from the reduction of accesses can be observed in the figure from the reduction in the energy consumption incurred by the *read* and *write* operations. Here, *read* and *write* operations already include operations for all data types (i.e., *ifmaps*, *ofmaps*, and *weights*). The savings are achieved because of the effective data partitioning and scheduling found through the design space exploration. Meanwhile, the energy savings from the reduction of row buffer conflicts and misses, are reflected by the reduction in the energy consumption incurred by the *activation* and *precharge* operations. These savings are mainly because of the effective DRAM mapping that exploits row buffer locality, to decrease the row buffer conflicts and misses. Furthermore, since the proposed mapping in ROMANet also optimizes the DRAM access latency, the standby energy is also reduced. We also observe that, for the AlexNet and the VGG-16, the FC layers consume more access energy compared to the CONV layers, as shown in Figure 3.24(a)-(b). The reason is that, in these networks, the FC layers have a large number of *weights* compared to the CONV layers. Hence, to access these weights, a proportional number of DRAM accesses are required, which leads to the high energy consumption. Meanwhile for the MobileNet, the FC layer does not dominate the DRAM access energy as the network only has a single FC layer whose number of weights and feature maps are relatively small (1024x1000 *weights*, 1024 *ifmaps*, and 1000 *ofmaps*).

#### 3.2.7.6 DRAM Access Energy Savings in Sparse Networks

To improve the energy efficiency of DNN-based systems, the network models are usually first passed through a compression framework for achieving a compact model that can be deployed in resource constraint mobile devices [HLL+18, HMD16]. Based on the study of different compression techniques, it has been observed that structured pruning techniques are more commonly employed due to their easy deployability using off-the-shelf libraries [HLL+18, YLP+17]. Therefore, to show the applicability of ROMANet for structurally pruned networks, we evaluate ROMANet for a sparse MobileNet, that is pruned using one of the state-of-the-art structured pruning techniques, i.e., AutoML for Model Compression (AMC) [HLL+18]. Figure 3.25 shows the DRAM access energy per inference for the sparse MobileNet. The figure shows that our ROMANet reduces the DRAM access energy by about 30% in the burst mode and by about 38% in the non-burst mode, when compared to the BWA. The main contributors to these savings are also the reductions in the number of DRAM accesses and the number of DRAM row buffer conflicts and misses. These results prove that our ROMANet can provide DRAM access energy savings even for sparse networks when compared to the state-of-the-art.

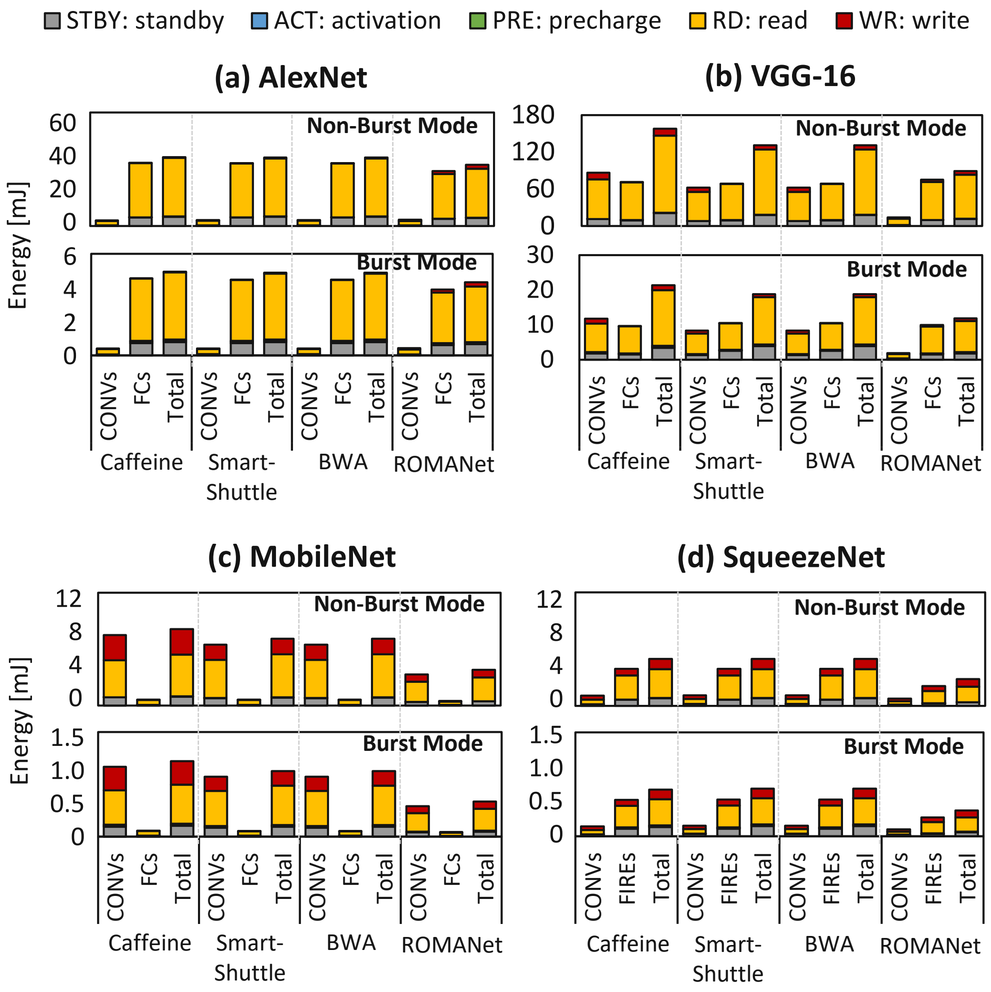


Figure 3.24: Results of DRAM access energy and the breakdown showing the contribution of different DRAM operations for (a) AlexNet, (b) VGG-16, (c) MobileNet, and (d) SqueezeNet. For AlexNet: CONVs refers to CONV1-CONV5 layers and FCs refers to FC1-FC3 layers; for VGG-16: CONVs refers to CONV1-CONV13 layers and FCs refers to FC1-FC3 layers; for MobileNet: CONVs refers to CONV1-CONV27 layers and FCs refers to FC1 layer; and for SqueezeNet: CONVs refers to CONV1-CONV2 layers and FIREs refers to FIRE1-FIRE8 layers.

### 3.2.7.7 Data Throughput Improvement for DRAM Accesses

Besides optimizing DRAM access energy, our ROMANet also improves the DRAM throughput in CNN accelerators. This can be observed from the experimental results presented in Figure 3.26. The figure shows that ROMANet improves the data throughput by about 10% in burst mode and 1.5% in non-burst mode, compared to the BWA. The main source of the improvement is that the proposed DRAM mapping uses bank-level parallelism to exploit DRAM multi-bank burst feature, thereby decreasing the possibility

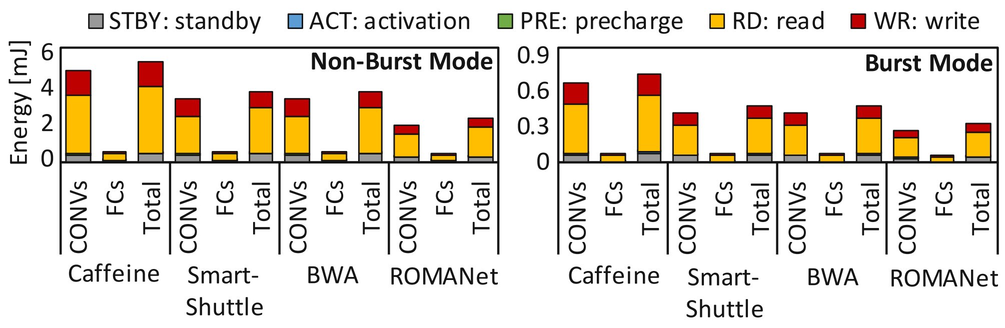


Figure 3.25: Results of the DRAM access energy and the breakdown showing the contribution of different DRAM operations in the sparse MobileNet. The CONVs refer to CONV1-CONV27 layers and the FCs refer to FC1 layer.

of facing row buffer conflicts. Figure 3.26 also shows that throughput improvement in the burst mode is typically higher than in the non-burst mode. The reason is that the non-burst mode requires a higher number of DRAM operations than the burst mode for accessing the same amount of data, thereby consuming higher DRAM access latency and decreasing the benefit of exploiting the DRAM multi-bank burst feature. Therefore, DRAM burst mode is preferred in the practical implementation of CNN accelerators. DRAM throughput improvement achieved by ROMANet is important as it enables fast data transfer between the off-chip DRAM and the on-chip SRAM buffers of CNN accelerators. This is beneficial for most of the existing CNN accelerators (such as [JYP$^+$17, LYL$^+$17]) since the obtained DRAM throughput can fulfill the bandwidth requirements to fully utilize the on-chip compute engines, and hence maximizing their processing potential.

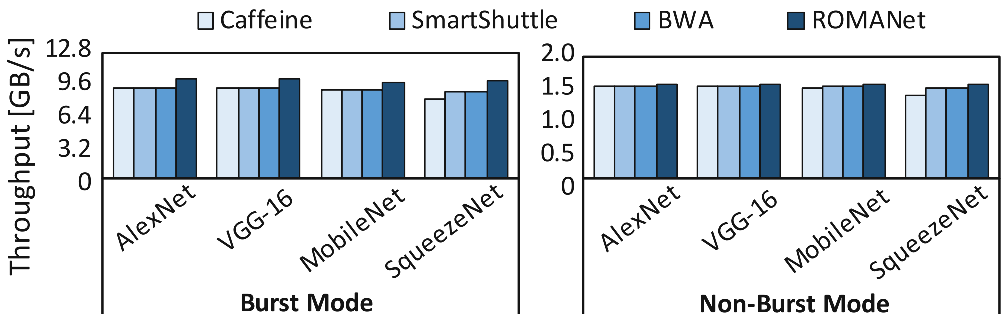


Figure 3.26: Results of the data throughput of DRAM accesses for the AlexNet, the VGG-16, the MobileNet, and the SqueezeNet using DDR3-1600 DRAM.

### 3.2.7.8 Additional Discussion

**DRAM access mode selection:** The DRAM access mode (i.e., burst mode or non-burst mode) is not defined by the amount of workload, but by the DRAM mapping. If the data are available in the subsequent DRAM columns, then the burst mode can speed up the data transfer.

**Impact of data precision on the DRAM access energy:** The data precision defines the bitwidth for each data value and the number of DRAM columns occupied. Therefore, the number of accesses required to fetch each value depends on the data precision, as shown in Figure 3.27(a). To achieve minimum DRAM accesses for each value, ROMANet employs the following mapping strategy: (1) the bits of each value are mapped to subsequent columns of a DRAM row, and (2) all bits of each value have to be placed to the same DRAM row. To quantify the impact of different bit precision settings, we perform experiments and the results are shown in Figure 3.27(b). The results show that the value with higher precision incurs higher DRAM access energy, due to a higher number of accesses.

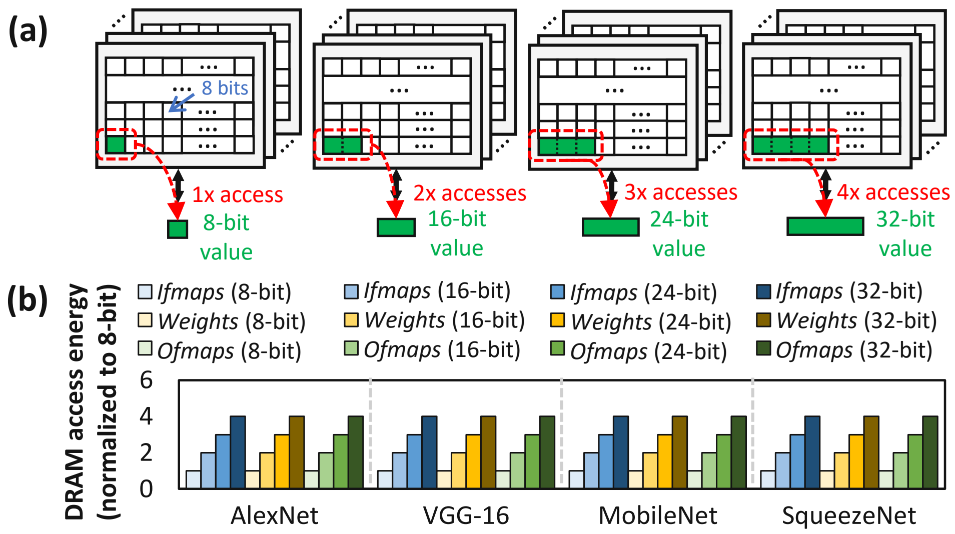


Figure 3.27: (a) The relation between different bit-precision settings of a single data with the number of DRAM accesses. (b) The DRAM access energy for different bit-precision settings for a DDR3-1600 2Gb x8 DRAM.

**On-chip buffer access latency and energy:** The total SRAM buffer access latency and energy are shown in Figure 3.28. Our ROMANet reduces the total buffer access latency and energy compared to the BWA, by 12% for the AlexNet, 34% for the VGG-16, 52% for the MobileNet, and 55% for the SqueezeNet. These reductions come from the decreased amount of data to be accessed from DRAM.

**Design-time overhead of the ROMANet:** The DSE of ROMANet is performed only once at compile time. It needs 2 hours for the AlexNet, 9 hours for the VGG-16, 2 hours for the MobileNet, and 5 hours for the SqueezeNet on Intel Core i7-7700 CPU @3.60GHz. Once the DSE is done, the data partitioning and scheduling information can be used by the existing software.

**Usability for different accelerators and applications:** Different CNN accelerators have different levels of flexibility to support different dataflows. For highly-flexible accelerators, ROMANet can be used directly, while for non-flexible ones, ROMANet can

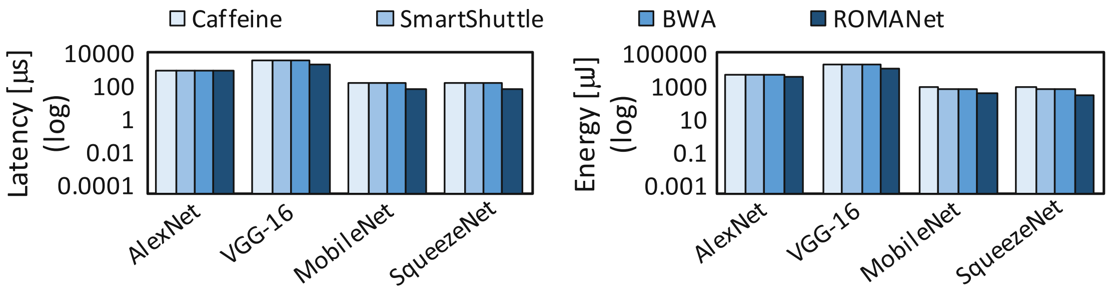


Figure 3.28: Results of the on-chip buffer access latency and energy. Each bar includes all SRAM buffers (i.e., *iBuff*, *wBuff*, and *oBuff*).

incorporate accelerator-specific constraints for the optimization process. Furthermore, in memory-bound applications, ROMANet can be employed directly. Meanwhile, in compute-bound applications, ROMANet can incorporate compute-specific constraints for the optimization process.

Note that the ROMANet methodology targets mainly the inference phase, which is particularly beneficial for memory-constrained embedded applications, such as for IoT and edge devices. Furthermore, our ROMANet can also be used for training to a certain extent, i.e., for forward computation, as its process is similar to inference. For backpropagation, the ROMANet needs enhancements to model the DRAM accesses for the required operations.

### 3.2.8 Summary of ROMANet Methodology

We propose a novel ROMANet methodology to reduce the DRAM access energy and to improve the DRAM throughput for CNN accelerators. It performs a design space exploration that finds the effective data partitioning and scheduling that offer minimum DRAM accesses. It also exploits the DRAM row buffer locality and the DRAM multi-bank burst feature by employing effective DRAM data mapping. The experimental results prove that the ROMANet can reduce the number of DRAM accesses, the row buffer conflicts and misses, as well as the operations that lead to significant DRAM access energy savings while improving the data throughput, as compared to the state-of-the-art works. Furthermore, our novel concepts would enable further studies on energy-efficient CNN accelerators.

## 3.3 High-Performance and Energy-Efficient DNNs using a Generalized DRAM Mapping

This section aims at addressing **Problem-2**, i.e., the solution for optimizing the DRAM energy-per-access and latency-per-access of different DRAM architectures for maximizing the efficiency gains of DNN accelerators.

### 3.3.1 Motivational Study

Although there are various types of commodity DRAMs (e.g., DDR3, DDR4, etc.), they have similar internal organization and operations [GLH+19], as discussed in Section 2.6. Therefore, they have similar trends regarding latency-per-access and energy-per-access. In commodity DRAMs, each request that goes to a bank can only access a single subarray at a time, although each bank is composed of multiple subarrays. This limits the capability of commodity DRAMs to offer lower access latency and energy. Recently, several DRAM architectures that offer subarray-level parallelism (SALP) in a bank, have been proposed in the literature. Three variants of SALP-based architectures (i.e., SALP-1, SALP-2, and SALP-MASA) are proposed in [KSL+12], as discussed in Section 2.6.4. Meanwhile, in another work [LKS+13], the long bitline in each DRAM subarray is split into two shorter segments (near and far segments from sense amplifier) using an isolation transistor (so-called TL-DRAM), as discussed in Section 2.6.4. Hence, the near segment of TL-DRAM can be accessed with the latency of a short bitline. To illustrate the characteristics of different DRAM architectures, we perform an experimental analysis to observe the DRAM latency-per-access and energy-per-access, and the experimental results are presented in Figure 3.29. Our observation results show that SALP and TL-DRAM architectures have the potential to further reduce the DRAM latency-per-access and energy-per-access as compared to commodity DRAMs, as they offer lower latency and/or energy in certain conditions; see ❶, ❷, and ❸ in Figure 3.29.

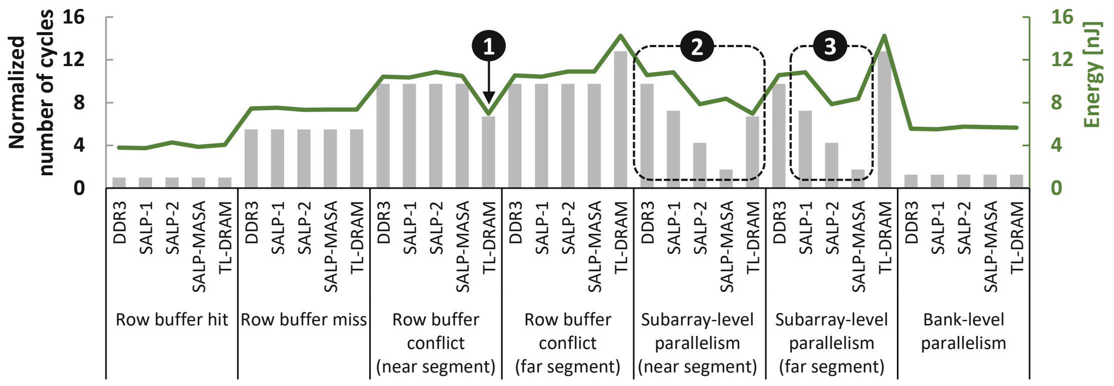


Figure 3.29: DRAM latency-per-access and energy-per-access for different access conditions (i.e., a row buffer hit, a row buffer miss, a row buffer conflict, subarray- and bank-level parallelism) in different DRAM architectures (DDR3, SALP-1, SALP-2, SALP-MASA, and TL-DRAM). Data are obtained from our experiments using state-of-the-art cycle-accurate DRAM simulators [KYM16, GYG+18] for DDR3-1600 2Gb x8 and SALP 2Gb x8 with 8 subarrays-per-bank.

### 3.3.2 Scientific Research Challenges

From previous observations, the energy efficiency of DRAM accesses for CNN accelerators can be improved by minimizing the DRAM latency-per-access and energy-per-access.

Hence, there is a need for *a generalized DRAM mapping policy* that can achieve maximum row buffer hits while exploiting the internal organization of different DRAM architectures, such as subarray-level and bank-level parallelism. Furthermore, to justify that the proposed DRAM mapping policy is applicable to different design choices, a design space exploration (DSE) is required. This DSE explores different DRAM mapping policies in different DRAM architectures with different data partitioning and scheduling schemes, to find the minimum energy-delay-product (EDP) of DRAM accesses. This EDP is used as a measure of the energy efficiency of a CNN accelerator. Therefore, *an analytical model to estimate the EDP of different DRAM mapping policies in DSE* is also needed.

***Required:*** *A methodology that leverages a generalized DRAM mapping policy to optimize DRAM energy-per-access and latency-per-access considering different DRAM architectures for CNN accelerators.*

### 3.3.3 Novel Contributions

To address the above challenges, we propose *PENDRAM, a novel methodology to enable high-Performance and Energy-efficient DNNs using a generalized DRAM data mapping policy* [PHS20] through the following key techniques; see an overview in Figure 3.30.

1. We propose **a generalized DRAM data mapping policy** that offers minimum EDP of DRAM accesses, for a given DRAM architecture, data partitioning, and scheduling scheme of CNN processing in a hardware accelerator. Our mapping policy orderly prioritizes maximizing row buffer hits, bank-level parallelism, and subarray-level parallelism in the near segment of the subarray.
2. We propose **a DSE algorithm** to find a DRAM mapping policy that offers minimum EDP, while considering different DRAM architectures, different data partitioning and scheduling schemes.
3. We propose **an analytical model for estimating EDP of different DRAM mapping policies**, which will be used in the DSE. Here, the EDP for each DRAM mapping policy is estimated by multiplying the number of DRAM accesses with the respective number of cycles and energy values.

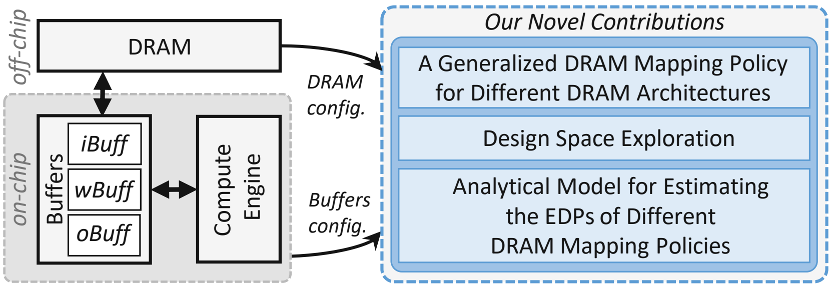

off-chip
DRAM
on-chip
Buffers
iBuff
wBuff
oBuff
Compute Engine
DRAM config.
Buffers config.
Our Novel Contributions
A Generalized DRAM Mapping Policy for Different DRAM Architectures
Design Space Exploration
Analytical Model for Estimating the EDPs of Different DRAM Mapping Policies

Figure 3.30: The overview of our novel contributions (shown in blue boxes). We consider separate on-chip buffers in a CNN accelerator for different data types: input buffer (*iBuff*) for *ifmaps*, weight buffer (*wBuff*) for *weights*, and output buffer (*oBuff*) for *ofmaps*.

### 3.3.4 PENDRAM Methodology

We propose PENDRAM methodology [PHS20] to optimize DRAM energy-per-access and latency-per-access for a given DRAM architecture, data partitioning, and scheduling scheme in CNN accelerators. Our PENDRAM methodology employs the following key techniques; see an overview in Figure 3.31.

- **A generalized DRAM data mapping policy (Section 3.3.4.1):** It aims at providing minimum EDP of DRAM accesses, for each given combination of DRAM architecture, data partitioning, and scheduling scheme. To achieve this, we leverage the characteristics of DRAM energy- and latency-per-access from different DRAM architectures to devise a mapping policy that orderly prioritizes maximizing row buffer hits, bank-level parallelism, and subarray-level parallelism in the near segment of the subarrays before moving to the far segments.
- **A DSE algorithm (Section 3.3.4.2)**: We perform DSE to ensure that the proposed DRAM mapping policy always achieves the minimum EDP. To do this, DSE investigates different combinations of DRAM mapping policies, DRAM architectures, as well as data partitioning and scheduling schemes, then evaluates the EDP for these combinations.
- To efficiently perform the DSE, we also propose and employ **an analytical model for EDP estimation of the DRAM mapping policies (Section 3.3.4.3)**. Our analytical model leverages the number of DRAM accesses and the characteristics of DRAM energy- and latency-per-access to calculate the EDP, i.e., by multiplying the number of DRAM accesses with the respective number of cycles and energy values.

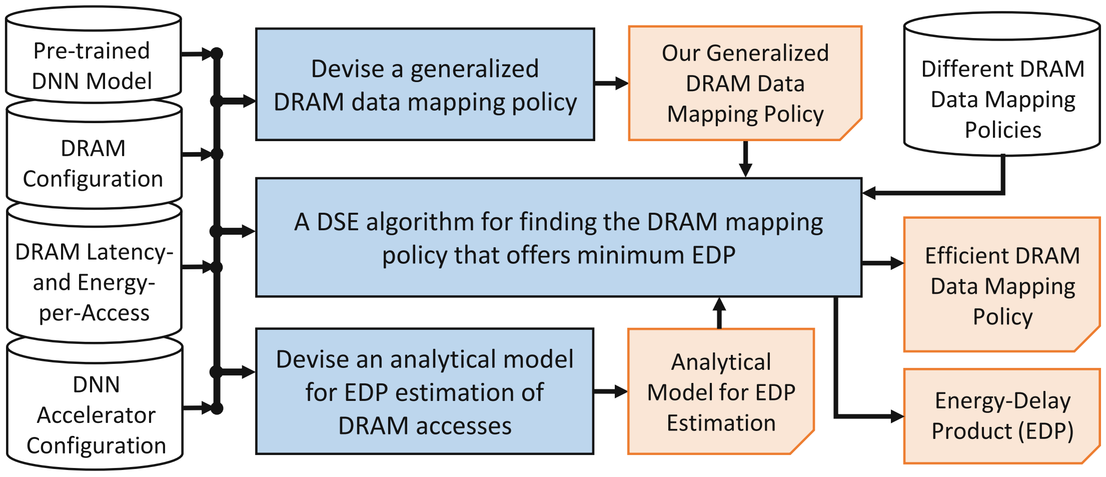
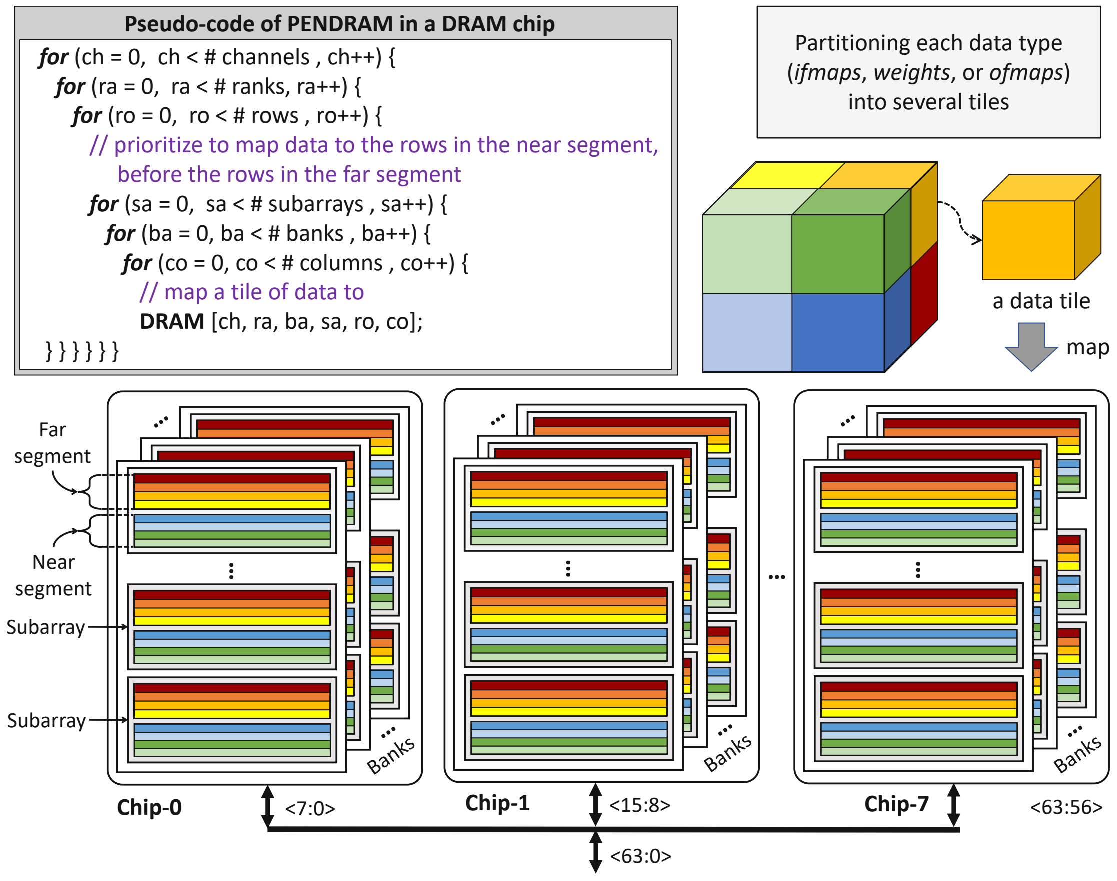

Pre-trained DNN Model
DRAM Configuration
DRAM Latency- and Energy- per-Access
DNN Accelerator Configuration
Devise a generalized DRAM data mapping policy
Our Generalized DRAM Data Mapping Policy
Different DRAM Data Mapping Policies
A DSE algorithm for finding the DRAM mapping policy that offers minimum EDP
Efficient DRAM Data Mapping Policy
Devise an analytical model for EDP estimation of DRAM accesses
Analytical Model for EDP Estimation
Energy-Delay Product (EDP)

Figure 3.31: An overview of the PENDRAM methodology. The novel contributions are highlighted in blue.

### 3.3.4.1 A Generalized DRAM Data Mapping Policy

The experimental results in Figure 3.29 show that different DRAM architectures have similar behavior in terms of latency-per-access and energy-per-access. Therefore, *we propose a generalized DRAM mapping policy for enabling high-performance and energy-efficient DRAM accesses in CNN accelerators (i.e., PENDRAM mapping)*. Its main idea is to orderly prioritize the data mapping that maximizes DRAM row buffer hit, bank-parallelism, and subarray-level parallelism in the near segments of the subarrays before mapping data to the far segments. The pseudo-code and physical representation of the proposed PENDRAM mapping policy in DRAM are illustrated in Figure 3.32.



**Pseudo-code of PENDRAM in a DRAM chip**

```
for (ch = 0,  ch < # channels , ch++) {
  for (ra = 0,  ra < # ranks, ra++) {
    for (ro = 0,  ro < # rows , ro++) {
      // prioritize to map data to the rows in the near segment,
         before the rows in the far segment
      for (sa = 0,  sa < # subarrays , sa++) {
        for (ba = 0, ba < # banks , ba++) {
          for (co = 0, co < # columns , co++) {
            // map a tile of data to
            DRAM [ch, ra, ba, sa, ro, co];
}}}}}}
```

Figure 3.32: Pseudo-code of PENDRAM mapping policy and its conceptual implementation in DRAM.

PENDRAM mapping considers tile-based partitioning in its mechanism, thereby it can be performed for each data tile using the following steps.

- **Step-1:** We prioritize mapping the data to an available row that is closer to the sense amplifier (i.e., target row) in the target subarray and the target bank, thereby prioritizing rows in the near segment over the far segment for data mapping. After the target row is identified, then we map the data to different columns in the same row to achieve maximum row buffer hits. If multiple chips are available within a rank, then this step can be performed in different chips in parallel for exploiting the chip-level parallelism. If some data remain but all columns in the same row of the target subarray and the target bank are fully filled, then we go to **Step-2**.

- **Step-2:** We map the remaining data to a different target bank in the same chip to exploit bank-level parallelism. If multiple chips are available, then this step can be performed in different chips in parallel. Then, we follow the mapping mechanism in **Step-1** to **Step-2** again until all data are mapped across all banks. If some data remain but all columns in the same row and the same subarray across all banks are fully filled, then we go to **Step-3**.
- **Step-3:** We map the remaining data to a different target subarray in the target bank to exploit subarray-level parallelism. If multiple chips are available, then this step can be performed in different chips in parallel. Then, we follow **Step-1** to **Step-3** again until all data are mapped. If some data remain but all columns in the same row across all subarrays and all banks are fully filled, then we go to **Step-4**.
- **Step-4:** We select a different row index as the target row from the target subarray and the target bank. Then, we follow **Step-1** to **Step-4** again until all data are mapped. If some data remain but all memory cells in a rank are fully filled, then we map the remaining data following **Step-1** to **Step-4** again until all data are mapped to a different rank and a different channel, respectively.

To illustrate that our PENDRAM mapping policy always achieves the minimum EDP of DRAM accesses in different possible conditions, we perform an extensive DSE. Our DSE investigates different combinations of DRAM mapping policies, different DRAM architectures (e.g., DDR3, SALP-1, SALP-2, SALP-MASA, and TL-DRAM), as well as different data partitioning and scheduling schemes, then estimates the EDP for these different combinations. *This DSE is important to corroborate that the best solution that provides the minimum EDP in each given combination is always the same as provided by our PENDRAM mapping policy.*

#### 3.3.4.2 DSE for Evaluating Different DRAM Mapping Policies

To evaluate the impact of different DRAM mapping policies and see the performance of our PENDRAM mapping as compared to others, we perform an extensive DSE. An overview of the DSE is shown in Figure 3.33 and its algorithm is presented in Algorithm 3. For each layer of a network, the DSE performs three key steps: (1) defining different sizes of data tiles and scheduling schemes, (2) defining different DRAM mapping policies, and (3) performing exploration to find a DRAM mapping policy that offers minimum EDP. The operational flow of the DSE is explained in the following points.

- **Step-①:** We define different sizes of data tiles for all data types (*ifmaps*, *weights*, and *ofmaps*), and different scheduling schemes. The tile sizes are defined by the step sizes in the outer loops of CNN processing in Figure 3.6(a). The tile sizes of *ifmaps*, *weights*,

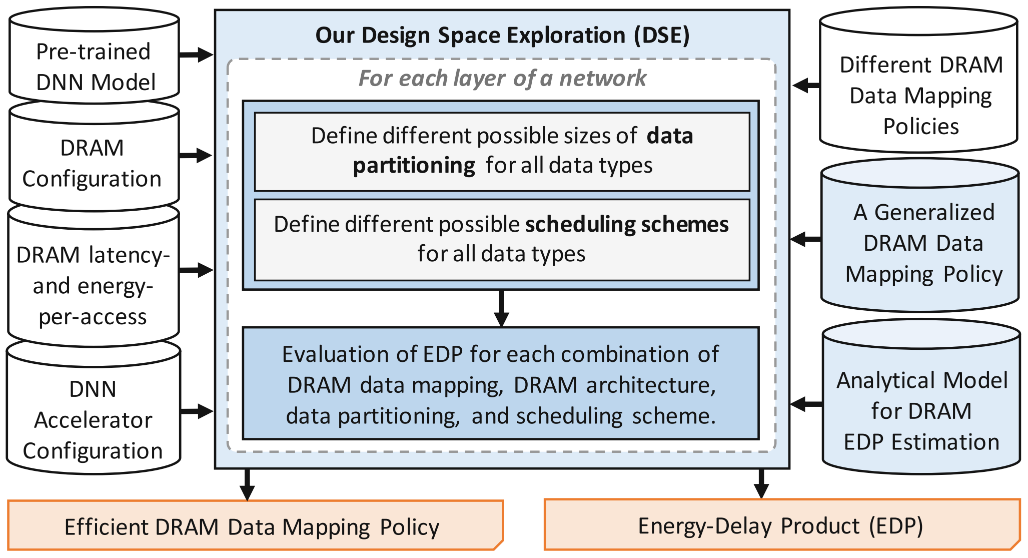

Pre-trained DNN Model
DRAM Configuration
DRAM latency- and energy-per-access
DNN Accelerator Configuration
Our Design Space Exploration (DSE)
For each layer of a network
Define different possible sizes of data partitioning for all data types
Define different possible scheduling schemes for all data types
Evaluation of EDP for each combination of DRAM data mapping, DRAM architecture, data partitioning, and scheduling scheme.
Different DRAM Data Mapping Policies
A Generalized DRAM Data Mapping Policy
Analytical Model for DRAM EDP Estimation
Efficient DRAM Data Mapping Policy
Energy-Delay Product (EDP)

Figure 3.33: The operational flow of our DSE.

and *ofmaps* have to fit in the corresponding buffers (*iBuff*, *wBuff*, and *oBuff*). Each combination of the tile sizes for all data types defines one possible partitioning, which will be considered in the DSE. The scheduling schemes are determined by the sequence of the outer loops of CNN processing in Figure 3.6(a). We consider four scheduling schemes, based on the reuse priority of the data type: *ifmaps-reuse*, *weight-reuse*, *ofmaps-reuse*, and *adaptive-reuse* scheduling. The *ifmaps-reuse* scheduling means that *ifmaps* data type will be maximally reused when the data are available in the on-chip buffer. A similar definition is also applied for *weights-reuse* and *ofmaps-reuse*. Meanwhile, the *adaptive-reuse* scheduling means that the reuse priority changes across different layers of a network, according to which one among *ifmaps-*/*weights-*/*ofmaps-reuse* scheduling that offers the minimum number of DRAM accesses.

- **Step-②:** We define different DRAM mapping policies, by determining the different orders of mapping loops to different columns, rows (including near and far segments), subarrays, and banks in the same DRAM chip. For DDR3, orders of mapping loops are the permutation of banks, rows, and columns in the same DRAM chip. For SALP architectures (SALP-1, SALP-2, and SALP-MASA), orders of mapping loops are the permutation of banks, subarrays, rows, and columns in the same DRAM chip. Meanwhile, for TL-DRAM, orders of mapping loops are the permutation of banks, subarrays, near and far segment rows, and columns in the same DRAM chip. Furthermore, we narrow down the design space by selecting the DRAM mapping policies that have the least frequent subsequent accesses to different *rows*, since it is the most expensive access in the same DRAM chip, for both latency and energy (as validated by Figure 3.29). Therefore, there are six mapping policies to be explored in the DSE, as presented in Table 3.3. Note, our proposed DRAM mapping policy is represented as *Mapping-3*.

**Algorithm 3** Pseudo-code of the proposed DSE algorithm

```
INPUT: (1) CNN configuration: number of layers ($L$);
    (2) Buffer size: for ifmaps ($iB$), for weights ($wB$), for ofmaps ($oB$);
    (3) Analytical models of EDP ($EDP$);
    (4) Data partitioning for ifmaps, weights, and ofmaps ($Partitioning$);
    (5) DRAM access scheduling ($Scheduling$);
    (6) DRAM mapping policies ($DRAMmaps$);
OUTPUT: (1) Efficient DRAM mapping ($map$);
    (2) Minimum EDP ($minEDP$);
    BEGIN
    Initialization:
 1: $T_p = P$;
 2: $T_q = Q$;
 3: $EDP[] = 0$;
 4: $minEDP[] = 0$;
    Process:
 5: for ($l = 1$ to $L$) do
 6:   for (each $Partitioning$) do
 7:     for (each $Scheduling$) do
 8:       for (each $DRAMmaps$) do
 9:         if (ifmaps tile $\leq iB$) and (weights tile $\leq wB$) and (ofmaps tile $\leq oB$) then
10:           Calculate $EDP[l]$;
11:           if (first loop) then
12:             $minEDP[l] = EDP[l]$;
13:           else if ($EDP[l] \leq minEDP[l]$) then
14:             $minEDP[l] = EDP[l]$;
15:             Save $map$, $minEDP$;
16: return (1) $map$; (2) $minEDP$;
    END
```

Table 3.3: Different DRAM mapping policies for the DSE. Note, our proposed DRAM mapping policy is represented as *Mapping-3*

| Mapping | Inner-most- to outer-most-loops |
|---|---|
| 1 | column, subarray, bank, row (from near segment to far segment) |
| 2 | subarray, column, bank, row (from near segment to far segment) |
| 3 | column, bank, subarray, row (from near segment to far segment) |
| 4 | bank, column, subarray, row (from near segment to far segment) |
| 5 | subarray, bank, column, row (from near segment to far segment) |
| 6 | bank, subarray, column, row (from near segment to far segment) |

- **Step-③:** We perform the DSE to find a DRAM mapping policy that offers minimum EDP, across different combinations of DRAM architectures, data partitioning, and scheduling schemes. The minimum EDP and the corresponding DRAM mapping are the outputs of the DSE, for the given combination of DRAM architecture, data partitioning, and scheduling scheme.

Note, the DSE also incorporates the characteristics of DRAM latency and energy for determining the EDP in the final results. For each layer of a network, the EDP value is obtained by multiplying the DRAM access energy and latency consumed by each combination of DRAM mapping policy, DRAM architecture, as well as configuration of data partitioning and scheduling scheme. Therefore, the DSE will be able to find the combination that offers minimum EDP for each layer of a network and minimum total EDP for a whole network.

#### 3.3.4.3 Analytical Model of EDP Estimation for DRAM Mapping Policies

Based on the proposed DSE, *the optimization problem is formulated to minimize the EDP of DRAM accesses for each layer of a network* and can be stated as

$$Objective: minimize\,(EDP_{layer}) \tag{3.18}$$

The EDP-per-layer $(EDP_{layer})$ is obtained by multiplying the energy-per-layer and latency-per-layer. The energy-per-layer is obtained by accumulating all energy values incurred from the DRAM accesses for all data tiles during the processing of a network layer. Meanwhile, the latency-per-layer is obtained by accumulating all latency values incurred from the DRAM accesses for all data tiles during the processing of a network layer. Note, the DRAM access latency and energy are calculated on the basis of DRAM accesses for each data tile since we consider the tile-based data partitioning approach. For each tile, the number of cycles required for DRAM accesses $(Ncycle_{tile})$ which represents the DRAM latency can be calculated using Equation 3.19, and the DRAM access energy $(Energy_{tile})$ can be calculated using Equation 3.20.

$$\begin{aligned} Ncycle_{tile} = & Naccess_{column} \cdot Ncycle_{column} + Naccess_{row\_near} \cdot Ncycle_{row\_near} + \\ & Naccess_{row\_far} \cdot Ncycle_{row\_far} + Naccess_{subarray} \cdot Ncycle_{subarray} + \\ & Naccess_{bank} \cdot Ncycle_{bank} \end{aligned} \tag{3.19}$$

$$\begin{aligned} Energy_{tile} = & Naccess_{column} \cdot Energy_{column} + Naccess_{row\_near} \cdot Energy_{row\_near} + \\ & Naccess_{row\_far} \cdot Energy_{row\_far} + Naccess_{subarray} \cdot Energy_{subarray} + \\ & Naccess_{bank} \cdot Energy_{bank} \end{aligned} \tag{3.20}$$

Term $Naccess_x$ denotes the number of accesses to a different DRAM-$x$. $Ncycle_x$ denotes the number of cycles incurred when accessing a different DRAM-$x$. Meanwhile, $Energy_x$ denotes the access energy incurred when accessing a different DRAM-$x$. For all terms, $x \in$ {column, row in the near segment, row in the far segment, subarray, bank}.

### 3.3.5 Evaluation Methodology

To evaluate our proposed methodology, we build the experimental setup, as shown in Figure 3.34. We use a cycle-accurate DRAM simulator, Ramulator [KYM16], to obtain the statistics of latency (i.e, DRAM cycle-per-access) for different DRAM access conditions (e.g., row buffer hits, row buffer misses, and row buffer conflicts, as well as subarray-level and bank-level parallelism) in different DRAM architectures. Meanwhile, to profile the DRAM energy-per-access, we use a real experiments-based DRAM energy simulator, VAMPIRE [GYG+18]. Information regarding the DRAM energy-per-access and cycle-per-access are used for the DSE, which considers different DRAM mapping policies, different DRAM architectures, as well as different data partitioning and scheduling schemes to find the DRAM mapping policy that offers minimum EDP. For the DSE, we consider a state-of-the-art TPU [JYP+17]-like CNN accelerator as shown in Figure 2.5(a), with a reduced size of on-chip buffers and MAC array engine, as specified in Table 3.4. We also employ separate on-chip buffers for different data types (*iBuff* for *ifmaps*, *wBuff* for *weights*, and *oBuff* for *ofmaps*). To represent different DRAM architectures, we consider DDR3, SALP architectures (SALP-1, SALP-2, and SALP-MASA), and TL-DRAM. For scheduling schemes, we use *ifmaps-reuse*, *weights-reuse*, *ofmaps-reuse*, and *adaptive-reuse* scheduling schemes. For DRAM mapping policies, we evaluate six mapping policies presented in Table 3.3. For inputs, we use the AlexNet [KSH12], the VGG-16 [SZ14], the MobileNet [HZC+17], and the SqueezeNet [IHM+16] for dense networks, as well as a the Sparse MobileNet that is achieved using the AutoML for Model Compression (AMC) [HLL+18] technique, while considering the ImageNet dataset [DDS+09].

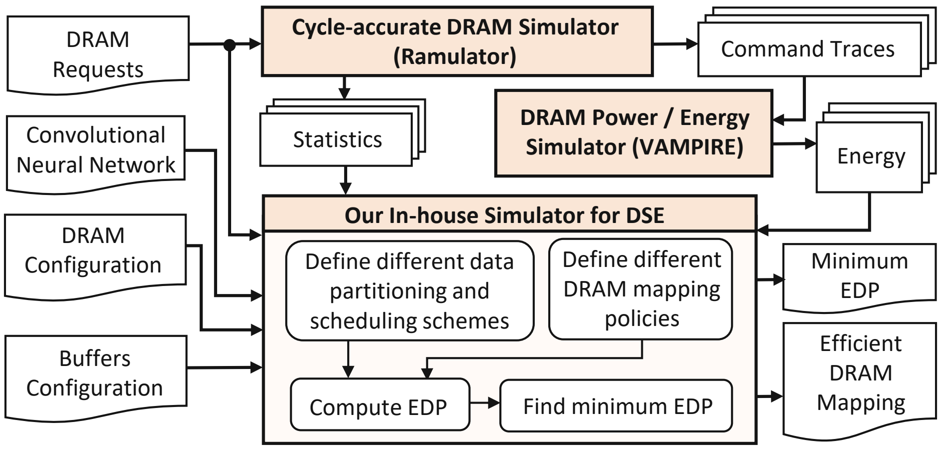

DRAM Requests
Convolutional Neural Network
DRAM Configuration
Buffers Configuration
Cycle-accurate DRAM Simulator (Ramulator)
Command Traces
Statistics
DRAM Power / Energy Simulator (VAMPIRE)
Energy
Our In-house Simulator for DSE
Define different data partitioning and scheduling schemes
Define different DRAM mapping policies
Compute EDP
Find minimum EDP
Minimum EDP
Efficient DRAM Mapping

Figure 3.34: Experimental setup and tool flow.

## 3.3.6 Results and Discussion

We evaluate the impact of different DRAM data mapping policies for a CNN accelerator across different DRAM architectures, data partitioning, and scheduling schemes. The experimental results are shown in Figure 3.35-3.39 for AlexNet, VGG-16, MobileNet, SqueezeNet, and Sparse MobileNet, respectively.

### 3.3.6.1 Comparisons of Different DRAM Mapping Policies

**Observation-①:** Our PENDRAM mapping policy (Mapping-3) achieves the lowest EDP across different layers of the network, different DRAM architectures, different

Table 3.4: Configuration of the CNN accelerator.

| Module | Description |
|---|---|
| CNN Processing Array | Size = 8 × 8 MACs |
| On-chip Buffers | *iBuff*: 64 KB, *wBuff*: 64 KB, *oBuff*: 64 KB |
| Memory Controller | Policy = open row, scheduler = FCFS |
| DDR3-1600 | Configuration: 2Gb x8<br>1 channel, 1 rank-per-channel<br>1 chip-per-rank, 8 banks-per-chip |
| SALPs | Configuration: 2Gb x8<br>1 channel, 1 rank-per-channel,<br>1 chip-per-rank, 8 banks-per-chip,<br>8 subarrays-per-bank |
| TL-DRAM | Configuration: 2Gb x8<br>1 channel, 1 rank-per-channel,<br>1 chip-per-rank, 8 banks-per-chip,<br>32 subarrays-per-bank,<br>64 rows/near segment, 960 rows/far segment |

scheduling schemes, and different networks. It indicates that the PENDRAM mapping is the most effective DRAM mapping policy as it always achieves the smallest EDP for each layer of networks across different possible conditions, thereby meeting the optimization objective described in Section 3.3.4.3. According to Table 3.3, our PENDRAM mapping (Mapping-3) *orderly prioritizes* mapping the data to (1) different columns in the same row, which leads to row buffer hits in DDR3, SALPs, and TL-DRAM; (2) different banks in the same chip, which exploits bank-level parallelism in DDR3, SALPs, and TL-DRAM; (3) different subarrays in the same bank with priority mapping to the near segment rows, which exploits subarray-level parallelism in SALPs and near segment accesses in TL-DRAM, but leads to row buffer conflicts in DDR3; and (4) different rows in the same subarray with priority mapping to the near segment rows, which exploits near segment accesses in TL-DRAM but leads to row buffer conflicts in DDR3 and SALPs. Following are detailed EDP improvements achieved by our PENDRAM mapping policy (Mapping-3) as compared to other mapping policies.

- For the AlexNet, our mapping improves the EDP by up to 96% in DDR3, 94% in SALP-1, 88% in SALP-2, 73% in SALP-MASA, and 96% in TL-DRAM.
- For the VGG-16, our mapping improves the EDP by up to 96% in DDR3, 94% in SALP-1, 89% in SALP-2, 77% in SALP-MASA, and 96% in TL-DRAM.
- For the MobileNet, our mapping improves the EDP by up to 96% in DDR3, 94% in SALP-1, 89% in SALP-2, 79% in SALP-MASA, and 95% in TL-DRAM.
- In the SqueezeNet, our mapping improves the EDP by up to 95% in DDR3, 93% in SALP-1, 90% in SALP-2, 81% in SALP-MASA, and 95% in TL-DRAM.
- For the Sparse MobileNet, our mapping improves the EDP by up to 96% in DDR3, 94% in SALP-1, 89% in SALP-2, 79% in SALP-MASA, and 95% in TL-DRAM.

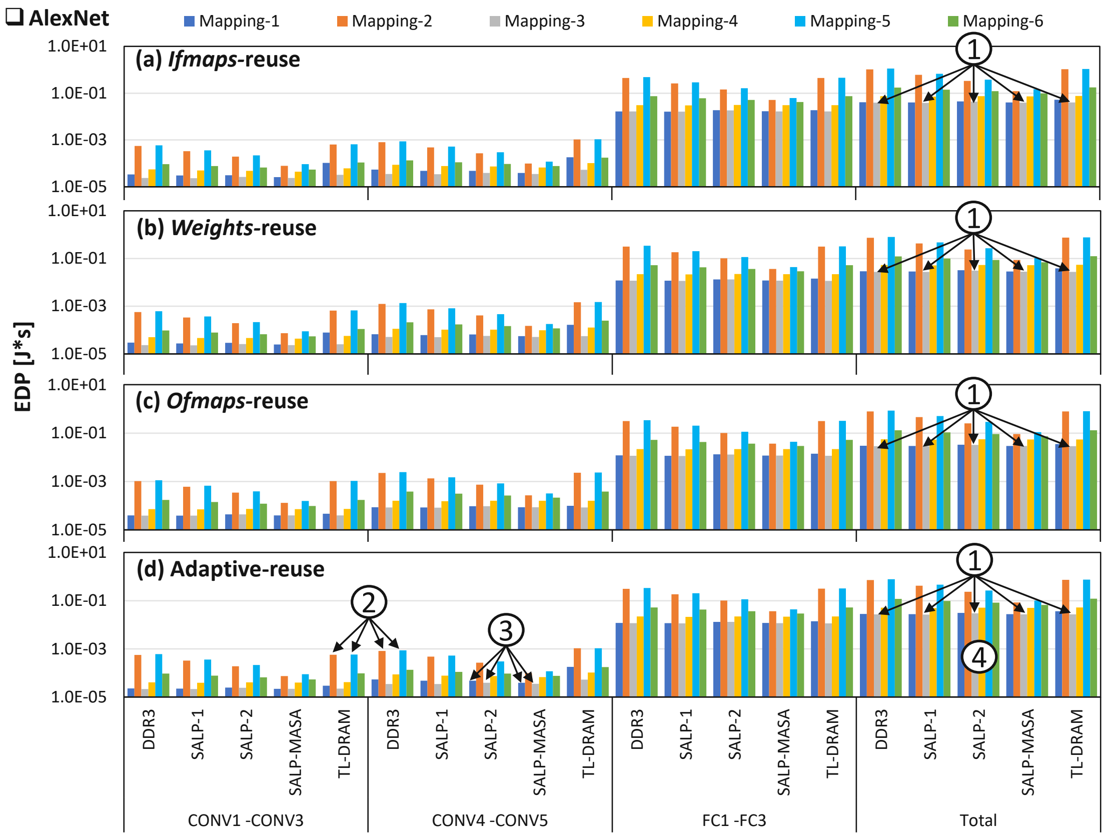


Figure 3.35: The EDP profiles of different DRAM mapping policies for the AlexNet across different DRAM architectures (i.e., DDR3, SALP-1, SALP-2, and SALP-MASA), while considering different data partitioning and scheduling schemes: (a) *ifmaps*-reuse, (b) *weights*-reuse, (c) *ofmaps*-reuse, and (d) adaptive-reuse.

*These results prove that our PENDRAM mapping policy is a generalized DRAM mapping policy that offers the lowest EDP for different design conditions.* Moreover, different DRAM access scheduling schemes can make use of the PENDRAM mapping, so that the CNN accelerators that employ different scheduling schemes can optimize their DRAM access latency and energy.

**Observation-②:** Mapping-2 and Mapping-5 obtain worse EDP values across different layers of the network, different DRAM architectures, and different scheduling schemes, than other mapping policies. The reason is that, Mapping-2 and Mapping-5 prioritize mapping the data across different subarrays in the same bank, which exploits subarray-level parallelism in SALPs, but leads to row buffer conflicts in DDR3 and may lead to far segment accesses in TL-DRAM. Consequently, these mapping policies incur higher EDP values as compared to other mapping policies that mainly exploit row buffer hits (Mapping-1 and Mapping-3) and bank-level parallelism (Mapping-4 and Mapping-6).

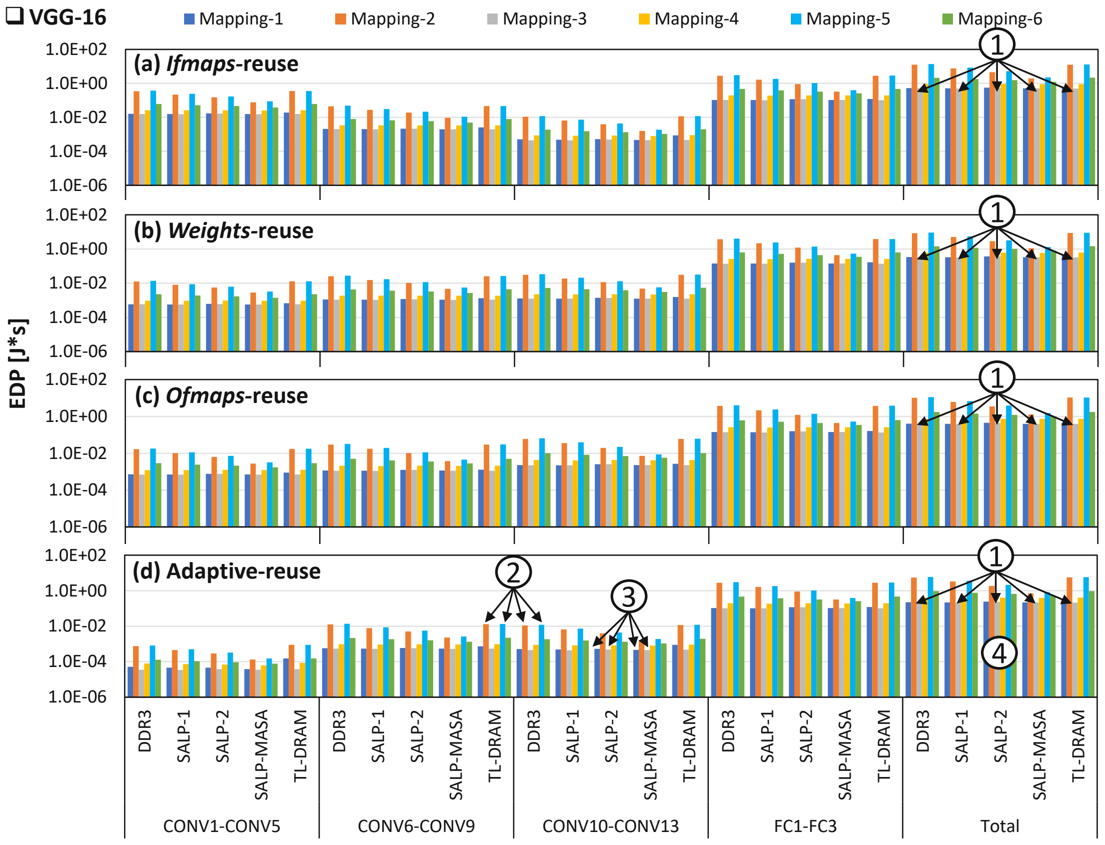


Figure 3.36: The EDP profiles of different DRAM mapping policies for the VGG-16 across different DRAM architectures (i.e., DDR3, SALP-1, SALP-2, and SALP-MASA), while considering different data partitioning and scheduling schemes: (a) *ifmaps*-reuse, (b) *weights*-reuse, (c) *ofmaps*-reuse, and (d) adaptive-reuse.

**Observation-③:** Mapping-1 and Mapping-3 obtain comparable EDP values across different layers of the network, different DRAM architectures, and different scheduling schemes. The reason is that, Mapping-1 and Mapping-3 prioritize mapping the data across different columns in the same row, which leads to row buffer hits in DDR3, SALPs, and TL-DRAM. The difference between these mapping policies comes when Mapping-1 prioritizes exploiting subarray-level parallelism over bank-level parallelism, while Mapping-3 is the opposite. From observation, bank-level parallelism incurs lower latency and energy than subarray-level parallelism as presented in Figure 3.29, thereby leading to lower EDP values for Mapping-3 (i.e., our PENDRAM mapping policy).

**Observation-④:** The adaptive-reuse scheduling scheme offers the lowest EDP for each layer of networks and the lowest total EDP for the given DNN models, as compared to other scheduling schemes. The reason is that, for each layer of networks, the adaptive-reuse scheduling scheme prioritizes keeping the data type that has the highest reuse factor on-chip for multiple operations, thereby maximizing the benefits of data reuse from different scheduling schemes (i.e., either *ifmaps*-reuse, *weights*-reuse, or *ofmaps*-reuse).

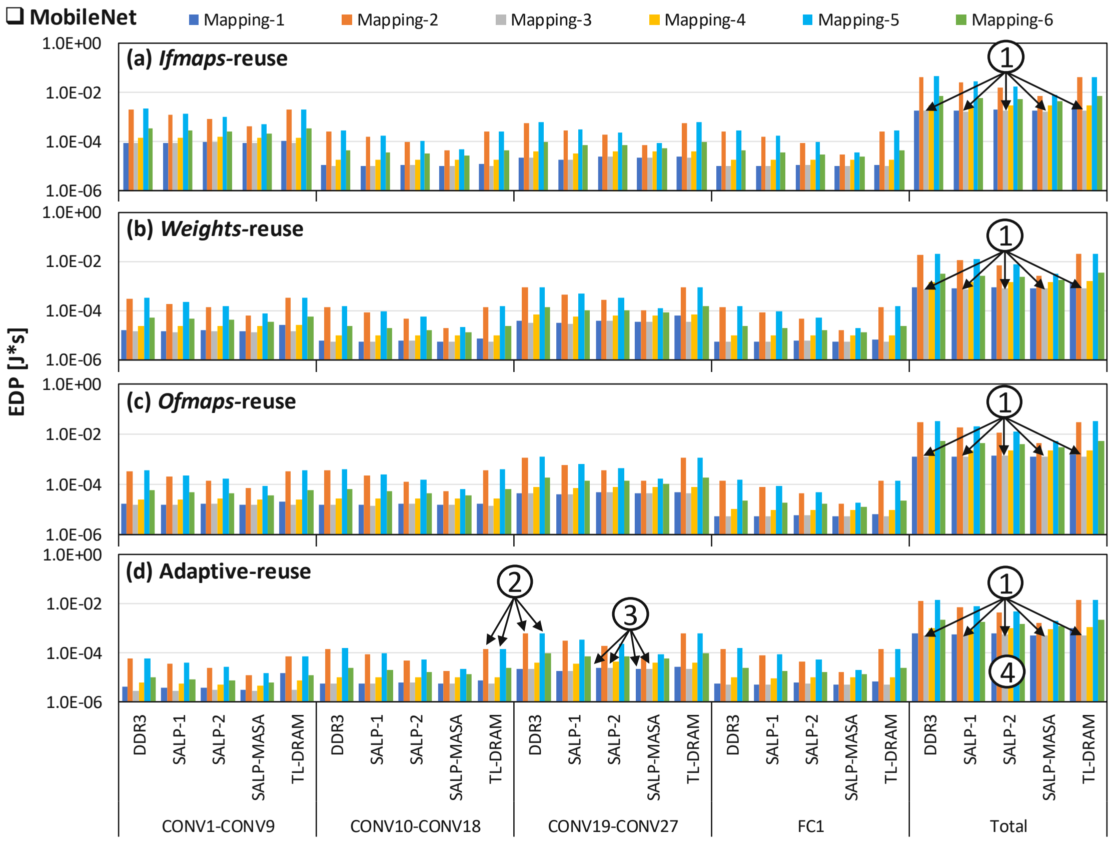


Figure 3.37: The EDP profiles of different DRAM mapping policies for the MobileNet across different DRAM architectures (i.e., DDR3, SALP-1, SALP-2, and SALP-MASA), while considering different data partitioning and scheduling schemes: (a) *ifmaps*-reuse, (b) *weights*-reuse, (c) *ofmaps*-reuse, and (d) adaptive-reuse.

### 3.3.6.2 Comparisons of Employing Different DRAM Architectures

In general, we observe that employing the SALP architectures (e.g., SALP-1, SALP-2, or SALP-MASA) can improve the EDP as compared to DDR3, across different DNN models. For instance, if we consider an adaptive-reuse scheduling scheme, the SALP architectures achieve EDP improvements by up to 88% for the AlexNet, by up to 87% for the VGG-16, by up to 86% for the MobileNet, by up to 81% for the SqueezeNet, and by up to 85% for the Sparse MobileNet. The EDP improvements offered by SALP architectures over DDR3 are most notable in Mapping-2 and Mapping-5 since these mapping policies prioritize mapping data across subarrays in the same bank, thereby exploiting subarray-level parallelism in SALP architectures but leading to row buffer conflicts in DDR3. Meanwhile, employing the TL-DRAM can also improve the EDP as compared to DDR3 across different DNN models. For instance, if we consider an adaptive-reuse scheduling scheme, TL-DRAM achieves EDP improvements by up to 4% for all investigated DNN models (i.e., AlexNet, VGG-16, MobileNet, SqueezeNet, and

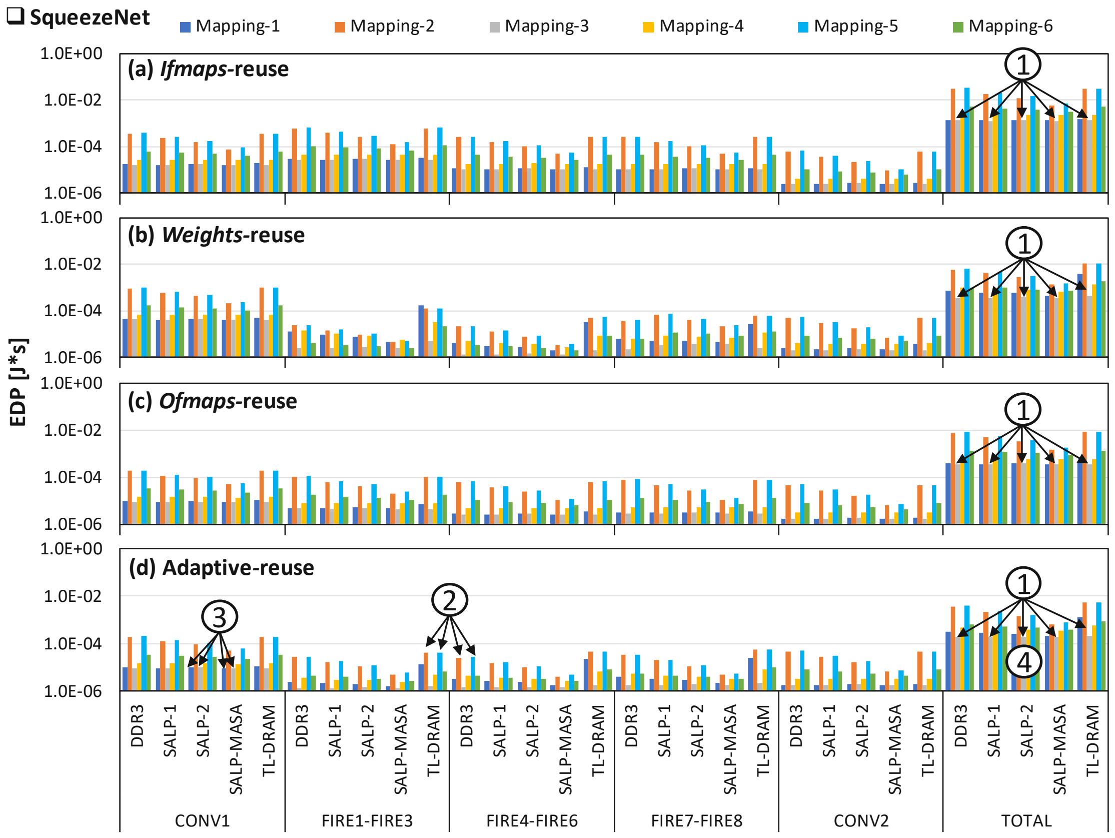


Figure 3.38: The EDP profiles of different DRAM mapping policies for the SqueezeNet across different DRAM architectures (i.e., DDR3, SALP-1, SALP-2, and SALP-MASA), while considering different data partitioning and scheduling schemes: (a) *ifmaps*-reuse, (b) *weights*-reuse, (c) *ofmaps*-reuse, and (d) adaptive-reuse.

Sparse MobileNet). The EDP improvements offered by TL-DRAM over DDR3 are most notable in Mapping-5 since this mapping policy prioritizes mapping data across subarrays in the same bank, thereby exploiting near-segment row accesses in TL-DRAM but leading to row buffer conflicts in DDR3. *Although these mapping policies provide improvements in novel DRAM architectures (i.e., SALPs and TL-DRAM), their EDP values for DRAM accesses are still higher than the EDP values achieved by our PENDRAM mapping policy (Mapping-3) across different scheduling schemes.* Therefore, these results lead to several observation points as follows.

- The EDP of employing different DRAM architectures may be different due to different DRAM access energy and latency profiles.
- Employing SALP architectures or TL-DRAM is beneficial for improving the energy efficiency of DRAM accesses in CNN accelerators, as long as an efficient mapping policy (i.e., our PENDRAM mapping) is employed to achieve the lowest EDP of DRAM accesses.

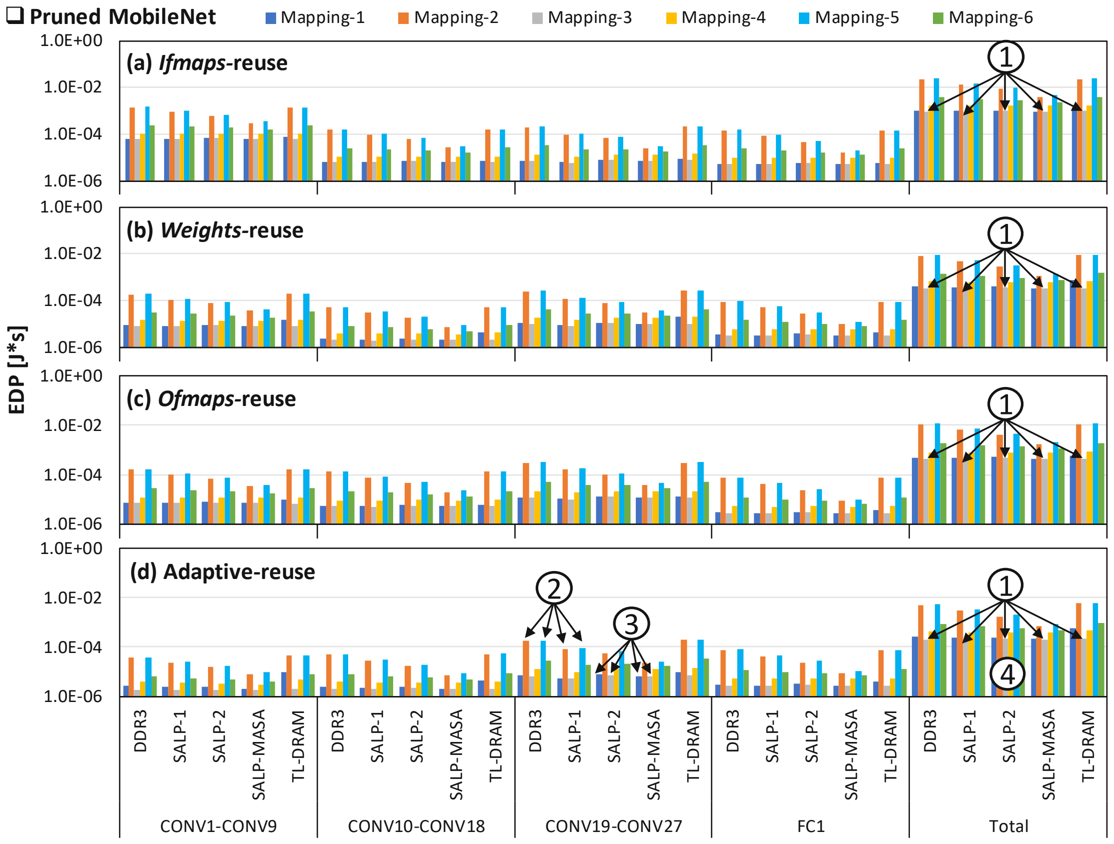


Figure 3.39: The EDP profiles of different DRAM mapping policies for the Sparse MobileNet across different DRAM architectures (i.e., DDR3, SALP-1, SALP-2, and SALP-MASA), while considering different data partitioning and scheduling schemes: (a) *ifmaps*-reuse, (b) *weights*-reuse, (c) *ofmaps*-reuse, and (d) adaptive-reuse.

- Since the internal organization of all DRAM architectures is similar (i.e., it is composed of channel, rank, chip, bank, subarray, row, and column), our PENDRAM methodology can be employed for all DRAM architectures to achieve energy-efficient processing of convolutional neural networks in CNN accelerators.

## 3.3.7 Summary of PENDRAM Methodology

We present the PENDRAM methodology which employs a generalized DRAM mapping policy that offers the lowest EDP of DRAM accesses for maximizing high-performance and energy-efficient CNN accelerators, as compared to other mapping policies. It is proven through an extensive design space exploration that evaluates the latency and energy of different mapping policies across different DRAM architectures (DDR3, SALP-1, SALP-2, SALP-MASA, and TL-DRAM), as well as different data partitioning and scheduling schemes. We expect that this work enables energy-efficient CNN accelerator designs and improves the DRAM access latency and energy for the existing CNN accelerators.

## 3.4 Summary of DRAM Optimization for DNN Systems

This chapter discusses our novel methodology for enabling energy-efficient DNN systems. It systematically addresses the targeted problems using our proposed optimization techniques. Specifically, it optimizes the dominant source of energy consumption for DNN inference (i.e., the DRAM access energy) through the reduction of the number of DRAM accesses and the DRAM energy-per-access. To do this, we perform a comprehensive design space exploration to find a combination of data partitioning and scheduling scheme that offers the minimum DRAM accesses. This exploration incorporates our analytical model which keeps the overlapping data on-chip for subsequent convolutional processes. Furthermore, we also propose a generalized DRAM data mapping to minimize DRAM energy-per-access and latency-per-access for DNN accelerators considering different DRAM design choices, including commodity DRAMs (e.g., DDR3) and novel DRAM architectures from the literature (e.g., SALP-1, SALP-2, SALP-MASA, and TL-DRAM). Our generalized DRAM data mapping policy is proven through a comprehensive design space exploration across different combinations of DRAM architectures, data partitioning, and scheduling schemes. In this manner, the DNN systems can perform inference in tight energy budgets, thereby making such systems suitable for many resource-constrained AI applications (e.g., Edge-AI and Smart CPS).

CHAPTER 4

# Energy-Efficient SNN Systems

This chapter discusses our novel methodology for achieving energy-efficient SNN systems. This chapter first identifies the problems for enabling energy-efficient SNN systems in both the training and inference phases in Section 4.1. To systematically address the research problems, we propose a novel methodology that employs our conjoint HW/SW-level design and optimization techniques for energy-efficient SNN systems. The proposed design flow is shown in Figure 4.1 and the details of novel contributions are described in the following sections in this chapter. Section 4.2 discusses a framework for optimizing the memory and energy requirements of SNN processing during both the training and inference phases. Section 4.3 further explores the quantization techniques for SNNs while considering different SNN parameters and quantization schemes. Afterward, Section 4.4 discusses a framework for employing approximate hardware (e.g., approximate DRAM) to substantially reduce the operational power/energy of SNN accelerators, while mitigating the negative impact of approximation on the accuracy. Apart from the development of SNNs through an offline training approach, we also propose a framework for enabling SNNs to continually adapt to diverse operational environments through our improved learning process, which is discussed in Section 4.5. Then, in Section 4.6, we further improve the learning process through learning rate enhancements and parameter adjustments to enable low-precision SNN processing for unsupervised continual learning scenarios.

## 4.1 Problem Identification

In an SNN architecture with unsupervised STDP learning rules depicted in Figure 2.12(a), each excitatory neuron is expected to recognize a class in the dataset, hence the connecting synapses from the same excitatory neuron have to learn the input features of a specific class. The existing works [HSS$^+$18, S$^+$17, PARR18, SPH$^+$19, HSS$^+$19] focus on improving the classification accuracy, but at the cost of a huge amount of additional computations, which leads to high energy and high memory footprint. For instance, the state-of-the-art work

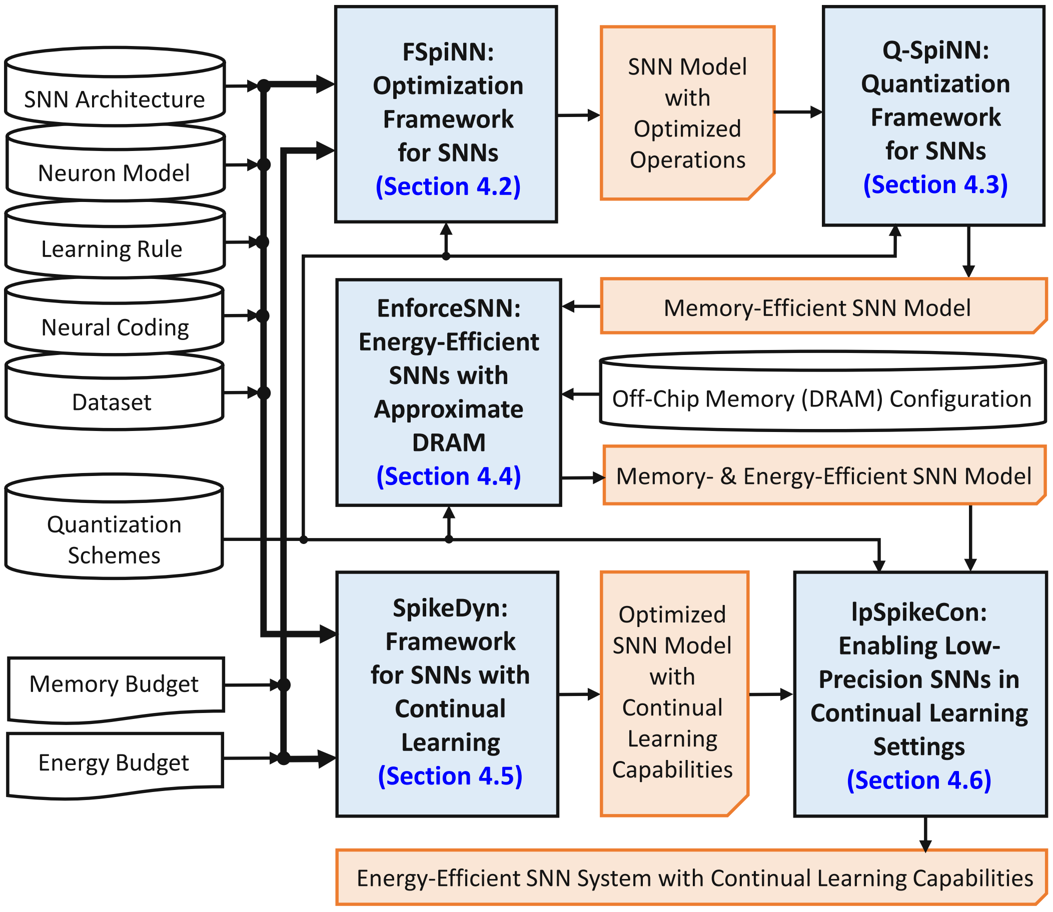

SNN Architecture
Neuron Model
Learning Rule
Neural Coding
Dataset
Quantization Schemes
Memory Budget
Energy Budget
FSpiNN: Optimization Framework for SNNs (Section 4.2)
SNN Model with Optimized Operations
Q-SpiNN: Quantization Framework for SNNs (Section 4.3)
EnforceSNN: Energy-Efficient SNNs with Approximate DRAM (Section 4.4)
Memory-Efficient SNN Model
Off-Chip Memory (DRAM) Configuration
Memory- & Energy-Efficient SNN Model
SpikeDyn: Framework for SNNs with Continual Learning (Section 4.5)
Optimized SNN Model with Continual Learning Capabilities
lpSpikeCon: Enabling Low-Precision SNNs in Continual Learning Settings (Section 4.6)
Energy-Efficient SNN System with Continual Learning Capabilities

Figure 4.1: An overview of the design flow of this chapter.

improves the effectiveness of the STDP-based learning by updating the weights every two or three postsynaptic spikes, to ensure that the update is essential [S+17]. This reduces the number of weight updates, but requires a 2-bit counter for each excitatory neuron to keep track of the number of postsynaptic spikes. Moreover, it also needs 200 neurons (i.e., 100 neurons for each excitatory and inhibitory layer), to achieve ∼75% accuracy in the MNIST digit classification[1], as shown in Figure 4.2(a). *Although all the existing techniques result in an improvement in accuracy, they incur high computational, memory, and energy costs. This is not desired for embedded applications with stringent constraints in computations, memory, and energy consumption (e.g., Edge-AI applications).*

**Problem-1:** *How can we optimize memory footprint and energy consumption of SNN processing for both the training and inference phases.*

[1]Unlike DNNs, the research for unsupervised learning-based SNNs is still in the early stage and mostly uses small datasets like MNIST and Fashion MNIST [DC15, HSS+18, S+17, PARR18, SPH+19, HSS+19, SLM19b]. We adopt the same test conditions used widely by the SNN research community [YAA+23].

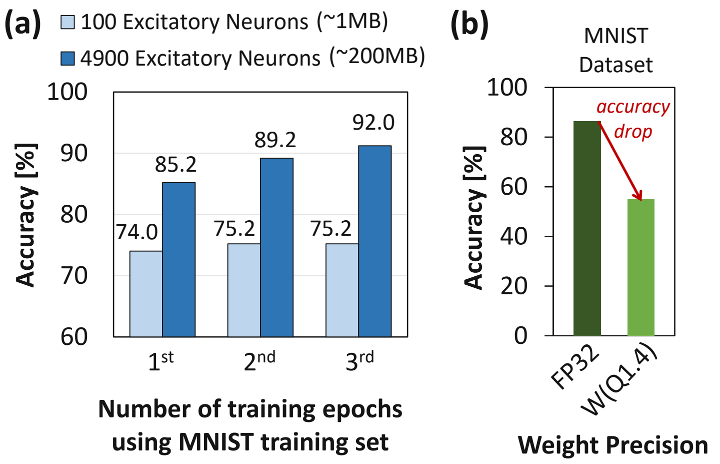


Figure 4.2: (a) A large-sized SNN typically achieves higher classification accuracy, e.g., the accuracy of an SNN with a total of 200 neurons (i.e., 100 excitatory and inhibitory neurons) is lower than an SNN with a total of 9800 neurons (i.e., 4900 excitatory and inhibitory neurons) on the MNIST dataset [LBBH98a]. (b) Accuracy of an SNN with 400 excitatory neurons with different weight precision levels on the MNIST dataset.

Previous works also proposed different methodologies to reduce the memory footprint of SNNs, such as (1) *reduction of SNN operations* via stochastic neuron operations [SVR17], neuron elimination [PS20], and weight pruning [RPR19]; and (2) *quantization* [RPR19, SLBS20, ZCG+20]. Among these techniques, quantization is a prominent one that incurs relatively low overhead, since it only needs to reduce the data precision. Besides memory saving, the reduced precision also leads to other advantages, e.g., faster computation and lower power/energy consumption. *However, reducing the precision of SNN parameters leads to accuracy degradation if it is not performed carefully due to the information loss*, as shown in Figure 4.2(b). The results show that a network with weight precision W($Q1.4$), i.e., 6-bit of fixed-point weights with 1 sign bit, 1 integer bit, and 4 fractional bits, suffers from an accuracy drop as compared to the 32-bit floating-point (FP32). The state-of-the-art works typically employed quantization to reduce the precision of the weights by directly using a specific quantization scheme, i.e., either the post-training quantization (PTQ) or the in-training quantization (ITQ) [RPR19, SLBS20, ZCG+20]. However, they have several drawbacks as they *do not* consider the following aspects.

- *Quantization for other SNN parameters* (e.g., neurons' membrane potential) that occupy a considerable amount of memory during the SNN processing [SVR17][RVG+17].
- *Exploration on different combinations of quantization approaches* (i.e., quantization schemes, precision levels, and rounding schemes) to find the SNN model that fulfills the targeted accuracy and achieves high memory saving.

Therefore, *the memory savings offered by the state-of-the-art works to meet the targeted accuracy are limited, thereby hindering the deployment of SNNs on the resource-constrained computing platforms.*

**Problem-2:** *How can we systematically employ the quantization on SNNs to maximize memory saving, while maintaining accuracy.*

Most of the SNN hardware platforms have relatively small on-chip memory, e.g., less than 100 MB [RVG+17, SVR17, FLLB19, FLB19]. Therefore, running an SNN model with a larger size than the on-chip memory of SNN hardware platforms, will require intensive access to the off-chip memory. Recent studies observed that a single access to the off-chip memory (i.e., DRAM) incurs significantly higher energy consumption than a single access to the on-chip memory (i.e., SRAM) [SCYE17, PHS21b]. Another work also identified that memory accesses dominate the energy consumption of SNN processing, incurring 50%-75% of the total system energy across different SNN hardware platforms, as shown in Figure 4.3. The reason is that, DRAM access energy is significantly higher than other SNN operations (e.g., neuron operations) [KSVR19]. This problem is even more critical for AI applications with stringent constraints (e.g., low-cost embedded devices with a small on-chip memory size) [SMWPH21], since it leads to even more intensive DRAM accesses. Consequently, this problem hinders SNN-based embedded systems from obtaining further efficiency gains.

To decrease the energy consumption of SNN inference, state-of-the-art works developed different optimization techniques, which can be loosely classified as the following.

- *Reduction of the SNN operations* through approximate neuron operations [SVR17], weight pruning [RPR19], and neuron removal [PS20]. These techniques decrease the number of DRAM accesses for the corresponding model parameters.
- *Quantization* by reducing the range of representable values for SNN parameters (e.g., weights) [RPR19, PS20, PS21a]. These techniques reduce the amount of SNN parameters (e.g., weights) to be stored in and fetched from DRAM.

These state-of-the-art works mainly aim at reducing the number of DRAM accesses, but do not optimize the DRAM energy-per-access and do not employ approximations in DRAM that provide an additional knob for obtaining high energy efficiency. Therefore, *optimization gains offered by the state-of-the-art works are sub-optimal, hindering the SNN inference systems from achieving the full potential of DRAM energy savings.*

**Problem-3:** *How can we substantially decrease the DRAM access energy of SNN hardware platforms for the SNN inference, while maintaining accuracy.*

Previous works have also explored different methodologies to build energy-efficient and unsupervised SNN systems, and most of them employed **offline training** [DC15, S+17, HSS+18, SPH+19, PS20]. However, the information learned by the offline-trained SNN system can be obsolete or may lead to low accuracy at run time under dynamically changing scenarios, as new data may have new features that should be learned online [PARR18, LDBK20, LLS+20, Ant19, vdVST20]. It becomes especially important for use

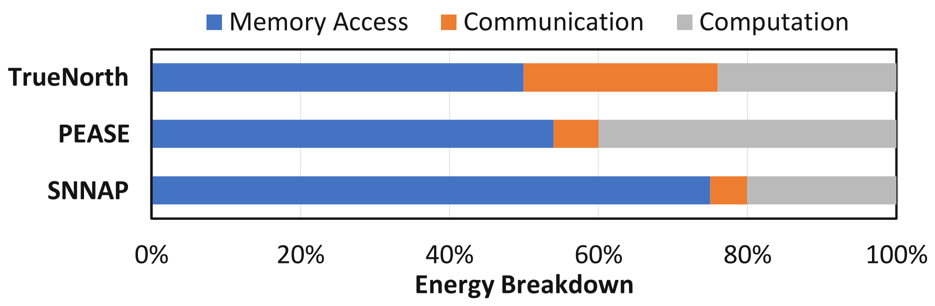


Figure 4.3: Energy breakdown of processing SNN in several hardware platforms, i.e., TrueNorth [ASC+15], PEASE [RVG+17], and SNNAP [SVR17] (adapted from the studies presented in [KSVR19]).

cases like IoT-Edge devices deployed in dynamically changing environments [LDBK20] and the autonomous agents (e.g., mobile robots, UAVs, and UGVs) in unknown terrains [LLS+20], as illustrated in Figure 4.4. New data that are gathered directly from such dynamic environments are usually unlabeled. Hence, an SNN-based system should employ unsupervised learning to process these data [PS20]. Moreover, new data are uncontrolled and their classes might not be randomly distributed, thereby making it difficult to learn different classes/tasks proportionally [AR16]. Therefore, the SNN system should employ **online training/learning** through *real-time continual learning*[2], while avoiding the undesired conditions, such as the following.

- The system learns information from the new data, but quickly forgets the previously learned ones (i.e., *catastrophic forgetting*) [MC89, CL18, PKP+19].
- The system mixes new information with the existing ones, thereby corrupting (polluting) the existing information [PARR18, AR16].
- The learning process needs a large number of weights and neuron parameters, and complex exponential calculations, thereby consuming high energy.

Previous works have tried to achieve continual learning through different techniques. The first category includes *the supervised learning techniques* that minimize the cost function in the learning process using data labels [KPR+17, LKJ+17, WBK06]. Hence, they cannot process unlabeled data which is required in the targeted problem. The second category includes *the unsupervised learning techniques* that perform learning using unlabeled data [PARR18, AR16, AR20]. However, the existing works suffer from spurious updates which lead to the sub-optimal accuracy, since they update the weights at each spike event, as observed in [S+17]. They also incur high energy consumption due to: (a) additional neurons [AR16, AR20]; (b) non-proportional quantities of training samples, i.e., samples from the earlier task are presented with larger quantities than later

[2]Continual learning is defined as the ability of a model to learn consecutive tasks (e.g., classes), while retaining information that have been learned [LDBK20, CL18, PKP+19]. Real-time means during the operational lifetime of the system.

tasks [PARR18]; and (c) large memory footprint and complex exponential calculations to achieve high accuracy [PARR18], i.e., a total of 800 neurons are needed to achieve 75% accuracy on the MNIST dataset[3]. Therefore, *the unsupervised continual learning capabilities offered by the state-of-the-art works are sub-optimal, hindering the SNN systems from enabling efficient mechanisms for adapting to dynamic environments.*

**Problem-4:** *How can we design lightweight and energy-efficient unsupervised continual learning for SNNs that adapts to dynamic environments for providing improved accuracy at run time.*

Furthermore, the existing works on unsupervised continual learning for SNNs typically employed high-precision weights (i.e., 32 bits) for both the training and inference phases to achieve high accuracy [PARR18, AR16, AR20], hence posing high memory and energy costs which hinder their efficient embedded implementations for battery-powered mobile autonomous agent applications. Toward this, quantization is a potential technique for efficiently reducing the memory footprint of SNNs, and thereby the energy consumption [PS20, PS21a]. However, *the impacts of weight quantization on the accuracy of unsupervised continual learning in SNN systems have not been explored yet.*

**Problem-5:** *How can we efficiently implement quantization for SNNs with unsupervised continual learning capabilities under tight memory constraints, while keeping the accuracy close to the baseline implementation.*

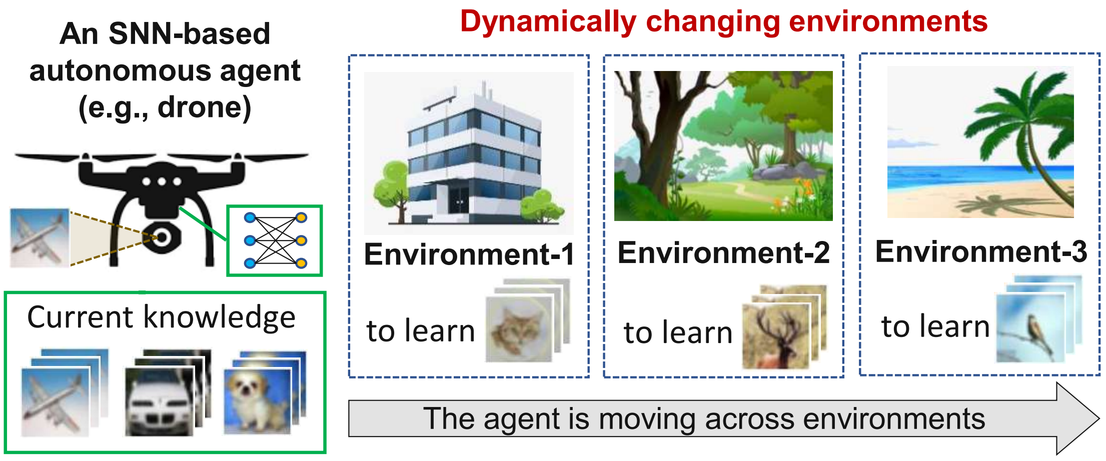

An SNN-based autonomous agent (e.g., drone)
Dynamically changing environments
Environment-1
to learn
Environment-2
to learn
Environment-3
to learn
Current knowledge
The agent is moving across environments

Figure 4.4: The SNN-based autonomous agent needs to perform training online using unsupervised continual learning strategies to update the knowledge, thereby adapting to dynamically changing environments in an efficient manner.

[3]Note, the research for the unsupervised continual learning in SNNs is still at an early stage and prominent SNN works mostly use the MNIST dataset [PARR18, AR16, AR20]. Therefore, we adopt the same test conditions as used widely by the SNN research community [YAA+23].

**Benefits:** The solution to these problems will enable *memory- and energy-efficient SNN systems with unsupervised continual learning capabilities that can adapt to diverse operational environments for energy-constrained embedded platforms and their applications for Edge-AI and Smart CPS*. For instance, mobile autonomous systems (e.g., robots, UGV, and UAVs) can employ such an SNN processing to perform inference (e.g., object recognition) as well as update the knowledge regularly (online learning) at run time.

**Proposed Solution:** To systematically address the above problems, we propose a comprehensive solution which is discussed in the following sections. Specifically, Problem-1, Problem-2, Problem-3, Problem-4, and Problem-5 are addressed in Section 4.2, Section 4.3, and Section 4.4, Section 4.5, and Section 4.6, respectively.

## 4.2 FSpiNN: An Optimization Framework for Memory-Efficient and Energy-Efficient SNNs

This section aims at addressing **Problem-1** with the solution for optimizing the memory footprint and the energy consumption of SNN processing for both the training and inference phases.

### 4.2.1 Motivational Study

In Figure 4.2(a), we observe that a large-sized SNN typically achieves higher classification accuracy and consumes a larger memory footprint. It shows that to achieve 92% accuracy for the MNIST digit classification, the SNN requires 9800 neurons (i.e, 4900 neurons for each excitatory and inhibitory layer) with 3 epochs of training, and consumes more than 200 MB of memory. On the other hand, most of the SNN hardware platforms have a limited size of on-chip memory (e.g., less than 100 MB) [ASC$^+$15, SVR17, RVG$^+$17, FLLB19], which makes running a large-sized network (whose size is larger than the on-chip memory) energy-consuming. The reason is that, this condition requires a high number of memory accesses, whose energy is typically higher than the compute operations [Hor14, CBM$^+$20b, PHS20]. Studies in [KSVR19] observed that the memory accesses are dominant, consuming about 50%-75% energy of SNN processing in different hardware platforms [ASC$^+$15, SVR17, RVG$^+$17], as shown in Figure 4.3.

We also observe that there are inefficient computations that hinder SNNs to achieve higher energy efficiency, which come from complex neuron and STDP operations. They require exponential calculations for computing the membrane and threshold potential decay, and the synaptic trace and weight dependence, respectively (see details in Section 2.3). Furthermore, there are ineffective STDP operations that come from spurious weight updates, which occur when the synapses of a neuron learn the overlapped features from different classes, thereby degrading the recognition capability of the neuron. This happens because the general STDP rule updates the synaptic weight on every presynaptic and postsynaptic spike, as discussed in Section 2.3.

### 4.2.2 Scientific Research Challenges

The high memory requirements in SNNs mainly come from a large number of parameters, such as synaptic weights and neuron parameters. Reducing these parameters may degrade the classification accuracy. Here, bitwidth quantization techniques may also be employed, but it can also lead to accuracy degradation. To overcome the limitations of the above optimization methods, parameter reduction should be done by identifying and eliminating the non-significant parameters. Furthermore, the STDP-based learning technique should be refined, so that the classification accuracy can be maintained/improved at minimal overheads.

***Required:*** *An optimization technique is required to reduce SNNs' memory and energy requirements for both training and inference processing, while maintaining classification accuracy, thereby enabling the SNN deployment on the memory- and energy-constrained embedded systems.*

### 4.2.3 Novel Contributions

To address the above challenges, we propose *FSpiNN, a novel optimization Framework for memory-efficient and energy-efficient Spiking Neural Networks for both the training and inference phases* [PS20], that employs the following key techniques (see an overview in Figure 4.5).

1. **Optimization of the neuron and STDP-based learning operations**. It replaces the inhibitory neurons with direct lateral inhibitory connections, reduces the presynaptic spike-based weight updates, and reduces the STDP complexity through the elimination of the exponential calculation in the weight dependence part.
2. **An algorithm for improving the STDP-based learning**. It minimizes the spurious weight updates through timestep-based operations, effectively updates the weights through an adaptive potentiation factor, and provides an effective competition among neurons through an adaptive inhibition.
3. **SNN quantization to compress the size of network parameters**. It employs a fixed-point format by rounding-to-the-nearest value, thereby providing a trade-off between the classification accuracy and the memory footprint.
4. **An algorithm to find the memory-aware and energy-aware SNN model**. It incorporates the memory and energy requirements in the optimization process, and employs a search algorithm to find the desired model.

### 4.2.4 FSpiNN Framework

FSpiNN framework [PS20] is an optimization framework for obtaining memory- and energy-efficient SNNs in both the training and inference phases while maintaining accuracy. FSpiNN employs the following key steps, and its flow is shown in Figure 4.6.

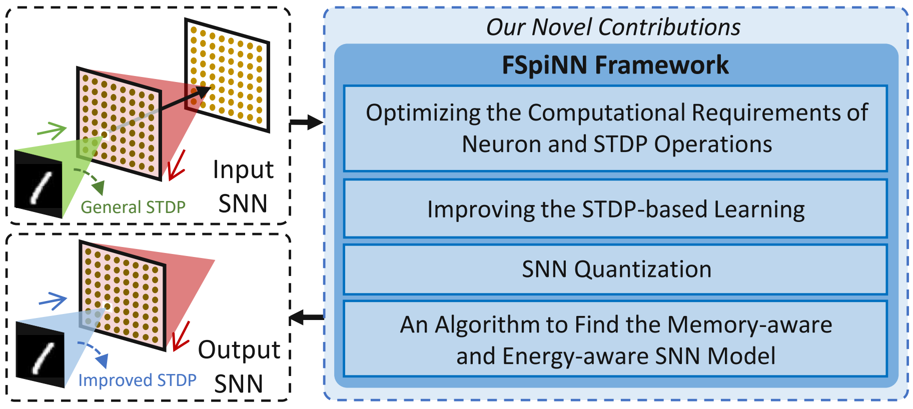

Input
SNN
General STDP
Output
SNN
Improved STDP
Our Novel Contributions
FSpiNN Framework
Optimizing the Computational Requirements of Neuron and STDP Operations
Improving the STDP-based Learning
SNN Quantization
An Algorithm to Find the Memory-aware and Energy-aware SNN Model

Figure 4.5: An overview of the FSpiNN framework. The novel contributions are highlighted in blue.

1. **Optimize the neuron and STDP-based learning operations (Section 4.2.4.1)** through the following means.

   - Reduce the number of neuron operations by replacing the inhibitory layer with the direct lateral inhibitory connections. It removes the inhibitory neurons and substitutes the function of spikes from the inhibitory neurons with spikes from the excitatory neurons.
   - Reduce the number of synaptic weight updates during STDP-based learning by eliminating the presynaptic spike-based weight updates. The updates happen only when the postsynaptic spikes occur, which indicates that the synapses learn the input features effectively.
   - Reduce the STDP complexity by fixing the weight dependence factor $\mu$ to 1, hence eliminating the complex exponential calculation.

2. **Improve the accuracy of STDP-based learning (Section 4.2.4.2)** through the following means.

   - Employ timestep-based synaptic weight updates to minimize the spurious weight updates that are induced by postsynaptic spikes, thereby ensuring that each update is essential.
   - Employ adaptive potentiation factor in STDP-based learning that makes use of the number of postsynaptic spikes to ensure how strong the potentiation should be applied in each weight update. It compensates for the loss of accuracy induced by the simplification of STDP operations.
   - Employ adaptive inhibition strength to proportionally provide competition among the excitatory neurons by applying a proper inhibition strength to other neurons. It is derived from an experimental analysis that investigates the accuracy of different inhibition strength values.

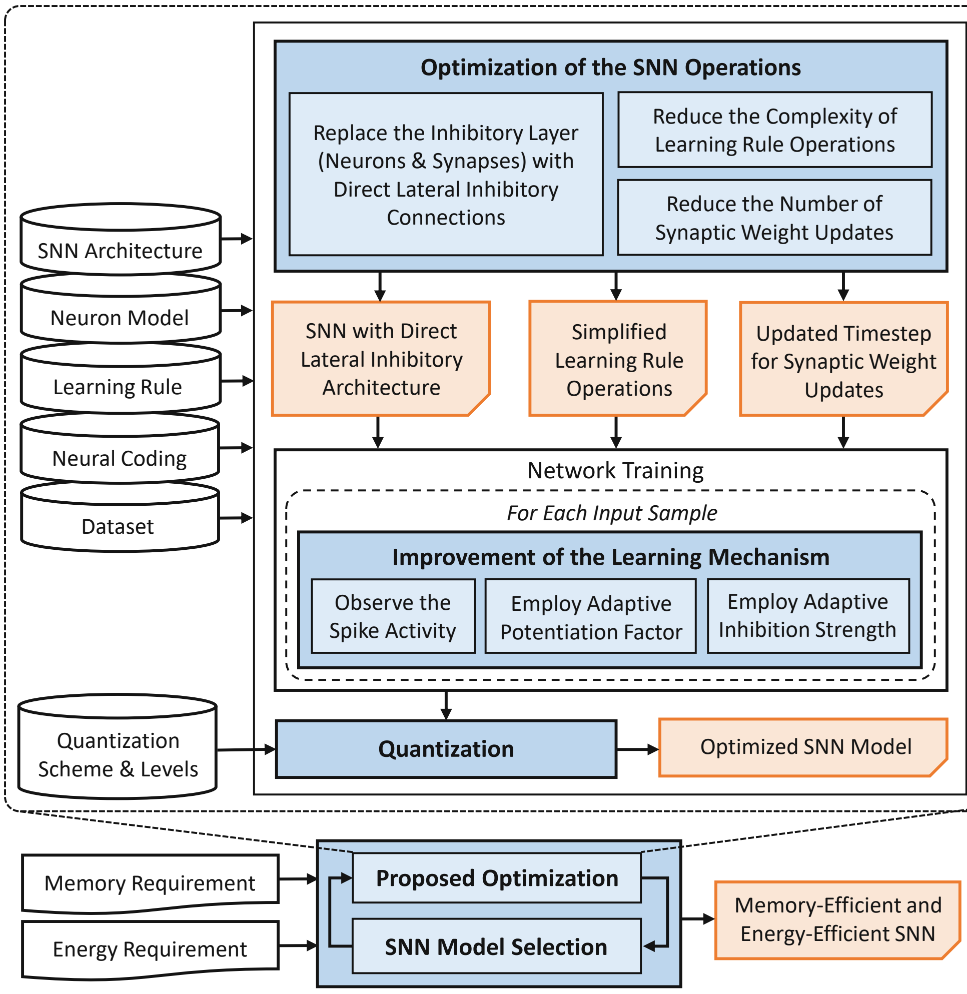

Optimization of the SNN Operations
Replace the Inhibitory Layer (Neurons & Synapses) with Direct Lateral Inhibitory Connections
Reduce the Complexity of Learning Rule Operations
Reduce the Number of Synaptic Weight Updates
SNN Architecture
Neuron Model
Learning Rule
Neural Coding
Dataset
SNN with Direct Lateral Inhibitory Architecture
Simplified Learning Rule Operations
Updated Timestep for Synaptic Weight Updates
Network Training
For Each Input Sample
Improvement of the Learning Mechanism
Observe the Spike Activity
Employ Adaptive Potentiation Factor
Employ Adaptive Inhibition Strength
Quantization Scheme & Levels
Quantization
Optimized SNN Model
Memory Requirement
Energy Requirement
Proposed Optimization
SNN Model Selection
Memory-Efficient and Energy-Efficient SNN

Figure 4.6: An overview of the FSpiNN framework. The novel contributions are highlighted in blue.

3. **Fixed-point SNN quantization (Section 4.2.4.3)** to compress the bitwidth of SNN parameters. It employs the rounding-to-the-nearest value technique, and explores the trade-off between the accuracy and memory costs for different quantization levels.

4. **A design space exploration (DSE) algorithm to find the SNN model (Section 4.2.4.4)** that fulfills the memory and energy budgets. It integrates a search algorithm with the proposed optimization to obtain a model that offers a good trade-off in memory footprint, energy consumption, and accuracy.

#### 4.2.4.1 Optimizing the Neuron and STDP Operations

**Reducing the number of neuron operations:** Our experiments in Figure 4.7 illustrate that the number of postsynaptic spikes generated from excitatory neurons is less than the presynaptic spikes. Therefore, the number of incoming spikes required to trigger an inhibitory neuron to generate a spike is less than the excitatory ones, and the inhibitory neuron typically has a smaller range of active membrane potential (between reset potential $v_{reset}$ and threshold potential $v_{th}$) compared to the excitatory ones. This indicates that the inhibitory neurons have different parameters from excitatory ones to be saved in memory. Hence, a large number of neurons utilized in the inhibitory layer will consume a considerable amount of memory and energy. Moreover, each inhibitory neuron needs to process only a small number of incoming spikes to generate the inhibition spike. Therefore, the use of inhibitory neurons could be optimized further to reduce memory and energy requirements.

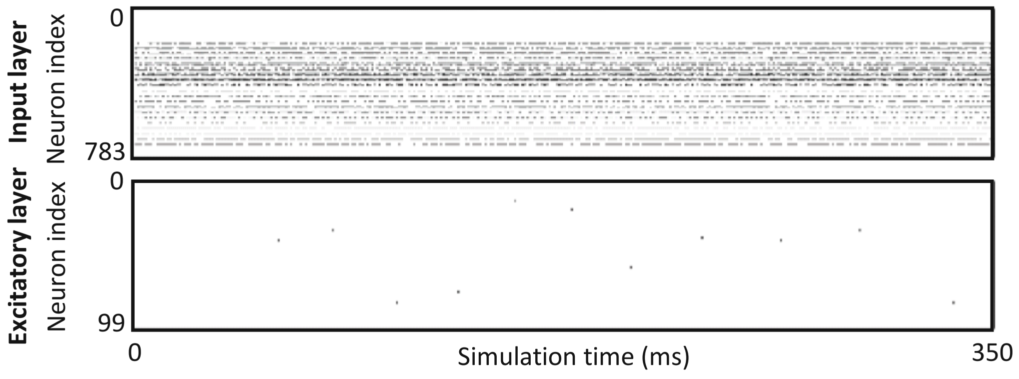


Figure 4.7: Illustration of the spike trains from the input and excitatory layers. It shows a significant difference between the number of spikes from the input layer (presynaptic spikes) and the excitatory layer (postsynaptic spikes).

*Proposed Optimization:* We propose to replace the inhibitory layer with direct lateral inhibitory connections to reduce the number of neurons in the network, thereby curtailing the neuron operations, as illustrated in Figure 4.8(a). In this manner, half of the total number of neurons are removed, and the function of spikes from the inhibitory neurons (to provide competition among excitatory neurons through a winner-takes-all mechanism) is substituted by the spikes from the excitatory neurons. Our experimental results in Figure 4.8(b) show that the lateral inhibitory connections have the potential to maintain accuracy, while having less resources than using the inhibitory layer. For instance, label-① in Figure 4.8(b) indicates that the SNN with a lateral inhibition can achieve a high accuracy faster than the SNN with an inhibitory layer, and then they converge to a comparable accuracy after more samples presented in the training phase. The reason is that, the lateral inhibition directly conveys spikes from an excitatory neuron to other neurons, hence the number of spikes is typically higher than the ones from the inhibitory layer. Therefore, the inhibition is stronger and it results in more diverse feature learning across neurons, thereby achieving high accuracy with a small number of training samples. This behavior is beneficial, especially when the SNN-based systems have only a small number of training samples.

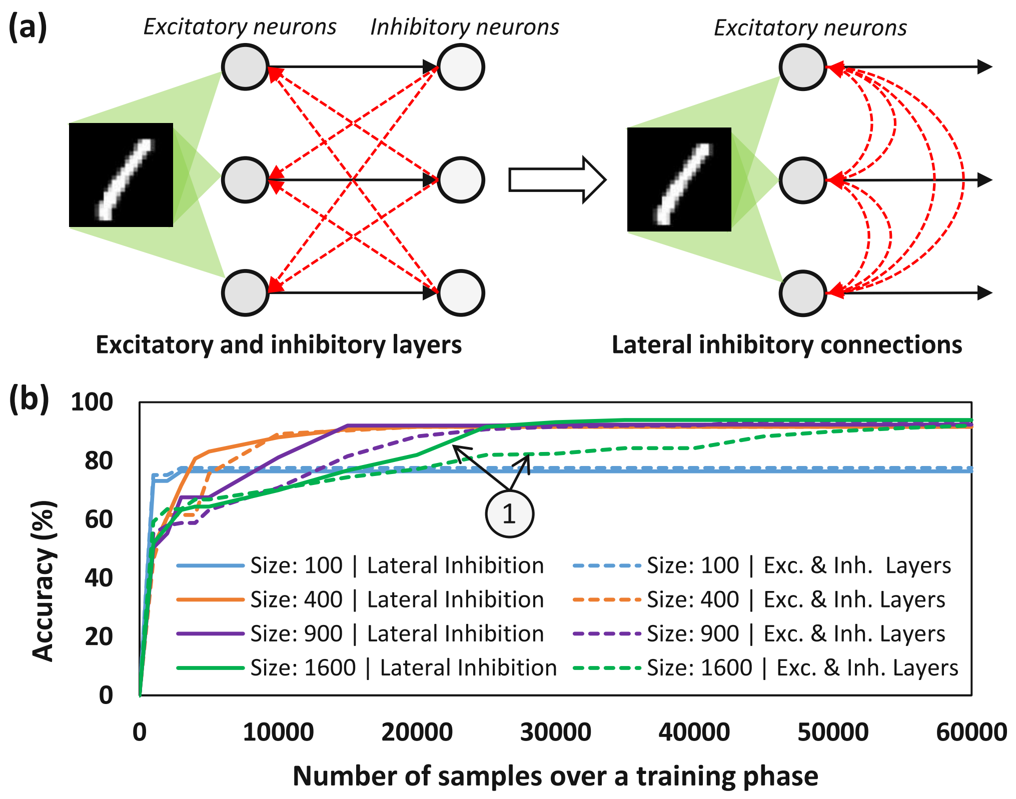


Figure 4.8: (a) Replacing the inhibitory layer with the direct lateral inhibitory connections (red-dashed lines), through which each excitatory neuron is connected to other excitatory neurons. (b) Impact of employing the direct lateral inhibitory connections on accuracy. This architecture offers comparable accuracy across different sizes of networks (i.e., 100, 400, 900, and 1600 excitatory neurons) as compared to employing the inhibitory layer.

**Reducing the number of STDP-based synaptic weight updates:** In the unsupervised SNNs, each neuron has to recognize features that belong to a specific class, so that each neuron can generate the highest number of spikes to represent its recognition category. To achieve this, the general STDP rule presented in Equation 2.15 updates the synaptic weight in every event of a presynaptic and postsynaptic spike. However, previous work observed that there are spurious weight updates which may decrease the accuracy of learning [S$^+$17]. The spurious updates are observed in two conditions: (i) when the neurons generate spikes unpredictably in the early phase of learning, due to the random weight initialization, and (ii) when a neuron generates spikes for patterns that belong to different classes, but share common features, thereby causing the synapses to learn the overlapped features from different classes. Therefore, the STDP-based weight updates that are induced by these presynaptic and postsynaptic spikes might not learn the input features effectively, and hence decreasing the recognition capability of the neuron and consuming energy. We exploit this observation in a new way to optimize the SNN computations, while preserving the classification accuracy.

*Proposed Optimization:* We propose to eliminate the presynaptic spike-based weight updates to reduce the spurious weight updates that are induced by the presynaptic spikes. Therefore, the learning will focus on the condition when postsynaptic spikes happen, which indicates that the connecting synapses effectively learn the input features. It also reduces the computational energy as the number of presynaptic spikes is higher than the postsynaptic ones, as shown in Figure 4.7.

**Reducing the STDP complexity:** The change in each synaptic weight ($\Delta w$) is updated using an STDP operation that requires complex exponential calculations for the synaptic trace and weight dependence parts (see Equation 2.15). We observe that the value of the weight dependence factor ($\mu$) is typically less than 1 [S$^+$17], which makes it expensive to compute. Therefore, the use of a weight dependence factor could be optimized to achieve further energy efficiency.

*Proposed Optimization:* We propose to fix the weight dependence factor $\mu$ to 1, thereby simplifying the computation of STDP operations. However, we observe that only fixing the weight dependence factor value may degrade the classification accuracy across different sizes of the network, as shown in Figure 4.9. Therefore, we propose a technique for improving the STDP-based learning (discussed in Section 4.2.4.2) to compensate for the loss of this $\mu$ simplification, and to maintain the accuracy.

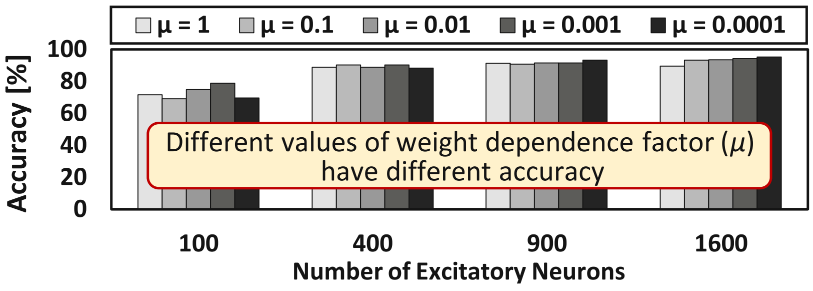


Figure 4.9: Impact of different values of weight dependence factor $\mu$ on accuracy, across different sizes of networks.

### 4.2.4.2 Improving the Accuracy of STDP-based Learning

We observe that for each input image, at least a single excitatory neuron is expected to recognize the input features, and then generate the highest number of spikes to represent a recognition of the corresponding class. Therefore, information regarding the number of postsynaptic spikes should be leveraged and used to improve accuracy.

*Proposed Solution:* We propose an algorithm to improve the accuracy of STDP-based learning by employing timestep-based synaptic weight updates, and adaptively determining the STDP potentiation factor ($k$) and the inhibition strength. Timestep-based synaptic weight updates aim to reduce the spurious weight updates, that are induced by the postsynaptic spikes. Hence, our technique updates the weight once within a defined timestep period, as long as at least there is a postsynaptic spike, as shown in Figure 4.10.

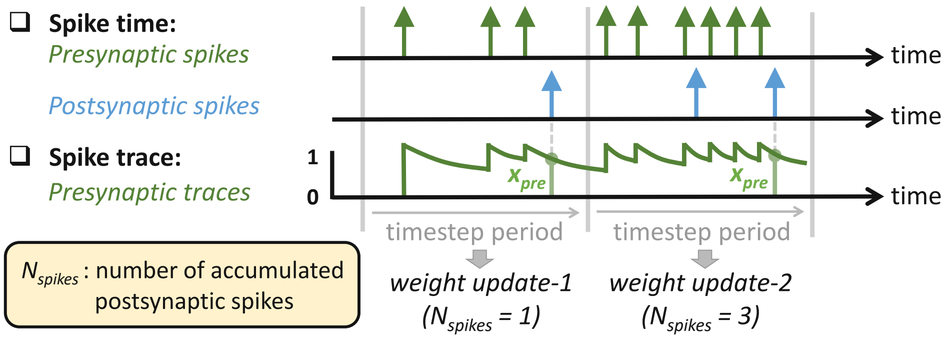


Figure 4.10: Overview of the timestep period, the synaptic weight updates, and the number of accumulated postsynaptic spikes ($N_{spikes}$) in our proposed technique.

We also propose an adaptive STDP potentiation factor $k$, which aims at determining how strong the potentiation should be in each weight update, by leveraging the number of postsynaptic spikes. To do this, our technique accumulates the number of postsynaptic spikes observed from the first time when the spike trains of an input sample (e.g., image) are presented to the network, until the time when a weight update is performed (denoted as $N_{spikes}$ in Figure 4.10). The number of postsynaptic spikes is used to determine the potentiation factor $k$, as formulated in Equation 4.1. Term $maxN_{spikes}$ denotes the maximum number of accumulated spikes, and $N_{spikes\_th}$ denotes the number of threshold spikes, which normalizes the value of $maxN_{spikes}$. Afterwards, the potentiation factor $k$ is used to compute the synaptic weight change $\Delta w$, as formulated in Equation 4.2. The synaptic weight update is conducted for the excitatory neuron that generates the highest number of postsynaptic spikes (i.e., the winning neuron). In this manner, the confidence level of learning is expected to increase over time when presenting the input spike trains.

$$k = \left\lceil \frac{maxN_{spikes}}{N_{spikes\_th}} \right\rceil \tag{4.1}$$

$$\Delta w = k \; \eta_{post} \; x_{pre} \; (w_m - w) \quad \text{on update time} \tag{4.2}$$

Furthermore, balancing the strength of excitatory and inhibitory synaptic connections is important since it makes the inhibition neither too strong, nor too weak. Too strong inhibition means that once the winning neuron is selected, it strongly prevents other excitatory neurons from firing, thereby dominating the recognition of input features (ineffective competition). Meanwhile, too weak inhibition means that it does not necessarily provide competition among the excitatory neurons, thereby giving no influence on the overall learning process (no competition). Previous studies in [DC15] observed that the ratio between the excitatory and inhibitory strengths have an important role to balance the learning process. Toward this, we perform an experimental analysis to investigate the accuracy in different inhibition strength conditions and different datasets to justify the generality of the effective ratio conclusion. The results are presented in Figure 4.11. Our analysis shows that when the inhibitory strength is too weak or too strong, the accuracy is sub-optimal. We observe that two comparable accuracy points are obtained using the ratio of 2x-4x. Therefore, we propose to use an adaptive inhibition strength that provides proper competition among the excitatory neurons, by applying an inhibition strength equal to 2x-4x of the excitatory strength.

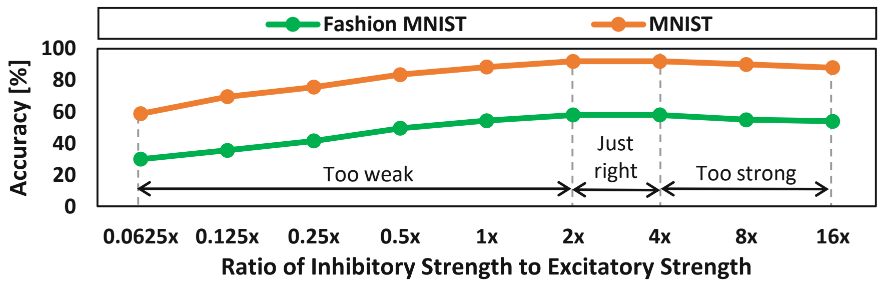


Figure 4.11: Impact of different ratio values between inhibitory and excitatory strengths when running the MNIST and Fashion MNIST datasets. When the ratio is too weak or too strong, the accuracy is sub-optimal.

Algorithm 4 synergistically employs the above-discussed techniques. For each excitatory neuron, the algorithm monitors whether a postsynaptic spike is generated. If so, the number of postsynaptic spikes is accumulated, and the corresponding presynaptic traces are recorded. Otherwise, no action is required. When the defined timestep period is reached, the algorithm identifies which excitatory neuron generates the highest number of spikes (the winning neuron). Once a winning neuron is identified, the connecting synapses to the winning neuron are updated with the synaptic weight change $\Delta w$.

#### 4.2.4.3 Fixed-Point Quantization for SNNs

It is a common practice to perform SNN processing using a single-precision floating-point operation to achieve a high classification accuracy. However, floating-point operations typically consume high memory and energy. To achieve memory- and energy-efficient SNN processing, it is more convenient to use a fixed-point format for neuron and STDP operations. However, quantizing a value implies a reduction of its representation capability, thereby decreasing the accuracy of the networks. Therefore, the quantization process should consider the trade-off between accuracy and memory requirements, to find acceptable quantization levels. In this manner, the users can select acceptable accuracy and memory to comply with the design specifications.

Toward this, our FSpiNN framework performs an exploration to investigate the impact of different quantization levels of SNN parameters (i.e., synaptic weights) on the accuracy, using the rounding-to-the-nearest value technique with the rounding half-up rule. This technique approximates the values that are halfway between two representable numbers by rounding them up. The fixed-point number can be written in $Qi.f$ format, as discussed in Section 2.5. The precision of the fixed-point format $\epsilon$ is defined as $\epsilon = 2^{-f}$, and it is used to define the quantized number $x_q$ as stated in Equation 4.3.

**Algorithm 4** Pseudo-code for improving the accuracy of STDP-based learning

```
INPUT: (1) Number of training dataset ($D_{train}$);
    (2) Simulation time for an input image ($t_{sim} = 350$);
    (3) Timestep ($t_{step} = 4$);
    (4) SNN parameters: number of excitatory neurons ($n_{exc}$), number of synapses-per-neuron
    ($n_{syn}$), number of accumulated postsynaptic spikes ($N_{spikes}$);
    (5) STDP parameters: learning rate ($\eta_{post} = 0.01$), max. weight value ($w_m = 1$), previous
    weight value ($w$), number of threshold spikes ($N_{spikes\_th} = 10$), potentiation factor ($k$);
    (6) Postsynaptic spike event ($spike_{post}$);
OUTPUT: Synaptic weight update ($\Delta w$);
BEGIN
    Initialization:
1:  $\Delta w[n_{exc}, n_{syn}] = zeros[n_{exc}, n_{syn}]$;
2:  $N_{spikes}[n_{exc}] = zeros[n_{exc}]$;
3:  $x_{pre} = zeros[n_{exc}, n_{syn}]$;
    Process:
4:  for ($d = 0$ to ($D_{train} - 1$)) do
5:      for ($t = 0$ to ($t_{sim} - 1$)) do
6:          for ($i = 0$ to ($n_{exc} - 1$)) do
7:              if $spike_{post}$ then
8:                  $N_{spikes}[i]$ += 1;
9:                  monitor $x_{pre}[i, :]$;
10:         if (($t \bmod t_{step}$) == 0) then
11:             $maxN_{spikes} = max(N_{spikes})$;
12:             $j \leftarrow index(max(N_{spikes}))$;
13:             $k = \lceil (maxN_{spikes}/N_{spikes\_th}) \rceil$;
14:             $\Delta w[j, :] = k\eta_{post} x_{pre}[j, :](w_m - w)$;
15: return $\Delta w$;
END
```

$$x_q = \left\lfloor x + \frac{\epsilon}{2} \right\rfloor \tag{4.3}$$

#### 4.2.4.4 DSE Algorithm for the Memory- and Energy-Aware SNN Model

To provide better applicability in many application scenarios, the proposed optimizations need to fulfill the given memory and energy requirements. Toward this, we also propose a DSE algorithm to find an SNN model whose memory and energy (for both the training and inference) are within the given memory and energy budgets, while maintaining high accuracy. The main idea is to incrementally increase the size of the SNN model (i.e., the number of excitatory neurons) and evaluate whether the currently investigated model satisfies the memory and energy budgets. If so, the DSE will evaluate whether the accuracy is better. If the accuracy is the same, the DSE will select the smaller model to keep the memory and energy consumption low. In this manner, our FSpiNN framework can support many applications where memory and energy are constrained. The pseudo-code of the algorithm is presented in Algorithm 5.

**Algorithm 5** Pseudo-code for the DSE algorithm

```
INPUT: (1) Memory requirement ($mem$);
    (2) Energy requirements: for training ($E_{train}$), for inference ($E_{inf}$);
    (3) SNN model ($model$): number of the excitatory neurons ($model.n_{exc}$), model size
    ($model.mem$), energy consumption for training ($model.E_{train}$), energy consumption for
    inference ($model.E_{inf}$), accuracy ($model.acc$);
    (4) Number of additional excitatory neurons ($n_{add}$);
OUTPUT: SNN model ($model$);
BEGIN
    Initialization:
 1: $model.n_{exc} = 0$;
 2: $model.size = 0$;
 3: $acc_{saved} = 0$;
    Process:
 4: while $model.size \leq mem_{req}$ do
 5:   if ($model.n_{exc} > 0$) then
 6:     perform training using Algorithm 4;
 7:     monitor $model.E_{train}$;
 8:     if ($model.E_{train} \leq E_{train}$) then
 9:       perform inference;
10:       monitor $model.E_{inf}$ and $model.acc$;
11:       if ($model.E_{inf} \leq E_{inf}$) and ($model.acc > acc_{saved}$) then
12:         $acc_{saved} = model.acc$;
13:         save $model$;
14:   $model.n_{exc} += n_{add}$;
15: return $model$;
END
```

### 4.2.5 Evaluation Methodology

Figure 4.12 shows the experimental setup for evaluating our proposed framework. We employ a Python-based SNN simulator [HSK+18] for evaluating the accuracy. We run the SNN simulations on different types of GPUs, namely Nvidia GeForce GTX 1060, GTX 1080 Ti, and RTX 2080 Ti (see Table 4.3), providing a wide range of compute and memory capabilities to show the scalability of our FSpiNN framework. We select the GTX 1060 and the GTX 1080 Ti as representatives of the Pascal architecture, which is used in the embedded GPUs, such as Nvidia Jetson TX2 [Nvie]. We also select the RTX 2080 Ti as representative of the Turing architecture, to provide variation in compute and memory capabilities. The same GPU architecture means the same technology and memory hierarchy. Hence, the experimental results can be used to estimate the relative energy-efficiency improvement obtained by our FSpiNN framework, as compared to the state-of-the-art works. From the simulations, we extract the size of the SNN model which represents the memory footprint. This information is used to evaluate memory savings. To estimate the energy, we adopt the approach in [HMD16]. We record the start and end time of the simulation to obtain the processing time, and employ the *nvidia-smi* utility to report the processing power, which are then used to estimate the energy consumption.

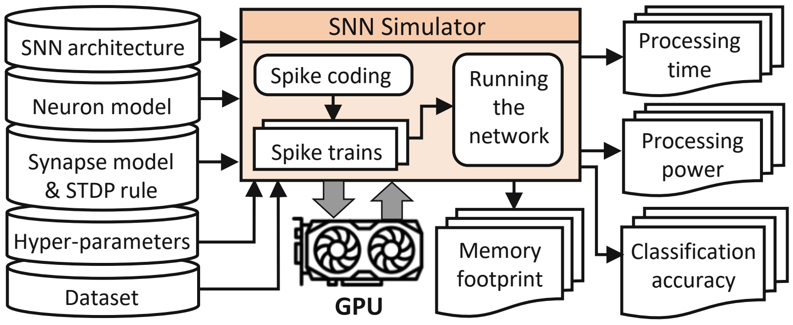


Figure 4.12: Experimental setup and tools flow.

Table 4.1: GPU specifications.

| Specifications | GTX 1060 [Nvia] | GTX 1080 Ti [Nvib] | RTX 2080 Ti [Nvic] |
|---|---|---|---|
| Architecture | Pascal | Pascal | Turing |
| CUDA cores | 1280 | 3584 | 4352 |
| Memory | 6GB GDDR5 | 11GB GDDR5X | 11GB GDDR6 |
| Interface width | 192 bit | 352 bit | 352 bit |
| Bandwidth | 8 Gbps | 11 Gbps | 14 Gbps |
| Power | 120 W | 250 W | 250 W |

**Datasets:** We use the MNIST [LBBH98a] and Fashion MNIST [XRV17] datasets, as they are widely used for evaluating the accuracy of SNNs [TGK$^+$19]. The MNIST represents a simple dataset, while the Fashion MNIST represents a more complex dataset [SLM19b]. Each dataset has 60,000 images for training and 10,000 images for testing, each having a dimension of 28x28 pixels.

**Input Encoding:** Every pixel of an image from the dataset is converted into a Poisson-distributed spike train whose firing rate is proportional to the intensity of the pixel. A higher-intensity pixel is converted into a higher number of spikes than a lower-intensity pixel. The spike train from each pixel is presented to the network for 350 ms duration.

**Classification:** In the training, the synaptic weight updates are performed without label information as it is unsupervised learning. Therefore, an additional mechanism is required to categorize the excitatory neurons for classification. The neurons are categorized based on their highest response to different classes over one presentation of the training set (1x epoch of training). Here, the labels are used to assign each neuron to a specific class. Afterward, the response of the class-assigned neurons is used to measure the accuracy.

**Comparisons:** We compare our proposed framework with two state-of-the-art designs, i.e., the general pair-wise weight dependence STDP-based SNN (baseline) [DC15], and the enhanced self-learning STDP-based SNN (SL-STDP) [S$^+$17]. The sizes of networks considered in the evaluation are the networks with different numbers of excitatory neurons: 100, 400, 900, 1600, 2500, 3600, and 4900. For conciseness, we refer them to as Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900, respectively. To provide fair comparisons, we recreate the baseline [DC15] and the SL-STDP [S$^+$17], and then simulate them using the same SNN simulator [HSK+18]. We also use the same approach for obtaining memory footprint and energy consumption. That is, we extract the size of the SNN model from simulations to evaluate the memory footprint, and we use the *nvidia-smi* utility to report the power and record the simulation time, which are then used to estimate the energy consumption. We also keep the hyper-parameter values the same for different sizes of networks. In particular, we use 1x epoch of training because the network will be trained with a full training set once. Moreover, an SNN model trained with 1x epoch of training is adopted by a wide range of researchers in the SNN community and considered a completely trained network [HSS+18, S+17, SPH+19, SSKR18].

### 4.2.6 Results and Discussion

#### 4.2.6.1 Maintaining Accuracy Comparable to State-of-the-Art

**Results for the MNIST Dataset:** Figure 4.13(a) shows the accuracy after 1x epoch of training. It shows that our FSpiNN maintains (and even improves in certain cases) the accuracy across different sizes of networks as compared to other designs (i.e., baseline and SL-STDP). Following are the detailed accuracy improvements achieved by FSpiNN from the baseline.

- Label-①: In Net100, FSpiNN achieves 13.2% improvement with 89.2% accuracy.
- Label-②: In Net400, FSpiNN achieves 7.2% improvement with 95.6% accuracy.
- Label-③: In Net900, FSpiNN achieves 2.4% improvement with 94.4% accuracy.
- Label-④: In Net1600, FSpiNN achieves 2.2% improvement with 95.2% accuracy.
- Label-⑤: In Net2500, FSpiNN achieves 0.8% improvement with 90% accuracy.
- Label-⑥: In Net3600, FSpiNN achieves 4.8% improvement with 92.8% accuracy.
- Label-⑦: In Net4900, FSpiNN achieves 2.4% improvement with 92.4% accuracy.

These results indicate that a larger network is harder to train. For instance, the accuracy achieved in Net100 and Net900 are 89.2% and 94.4% respectively, but the accuracy improvements in Net100 and Net900 are 13.2% and 2.4% respectively. The reason is that, a larger network has more synapses to train for learning the input features, thereby requiring more careful training (e.g., hyper-parameter tuning). This condition may cause the accuracy of the larger networks lower than the smaller ones in certain cases. For instance, the accuracy achieved in Net4900 is 92.4%, which is lower than the accuracy in Net900 (i.e., 94.4%). Furthermore, Figure 4.13(b) shows the synaptic weights and its classification matrix, and Figure 4.13(c) shows the confusion matrix for Net400. These results show the common confusions happen when identifying between digits 3 and 8, 4 and 9, etc. The reason is that, the connecting synapses from the same neuron learn the common features (shape) from these classes, hence the same neuron generates the highest number of spikes for different classes, that results in more frequent false classification.

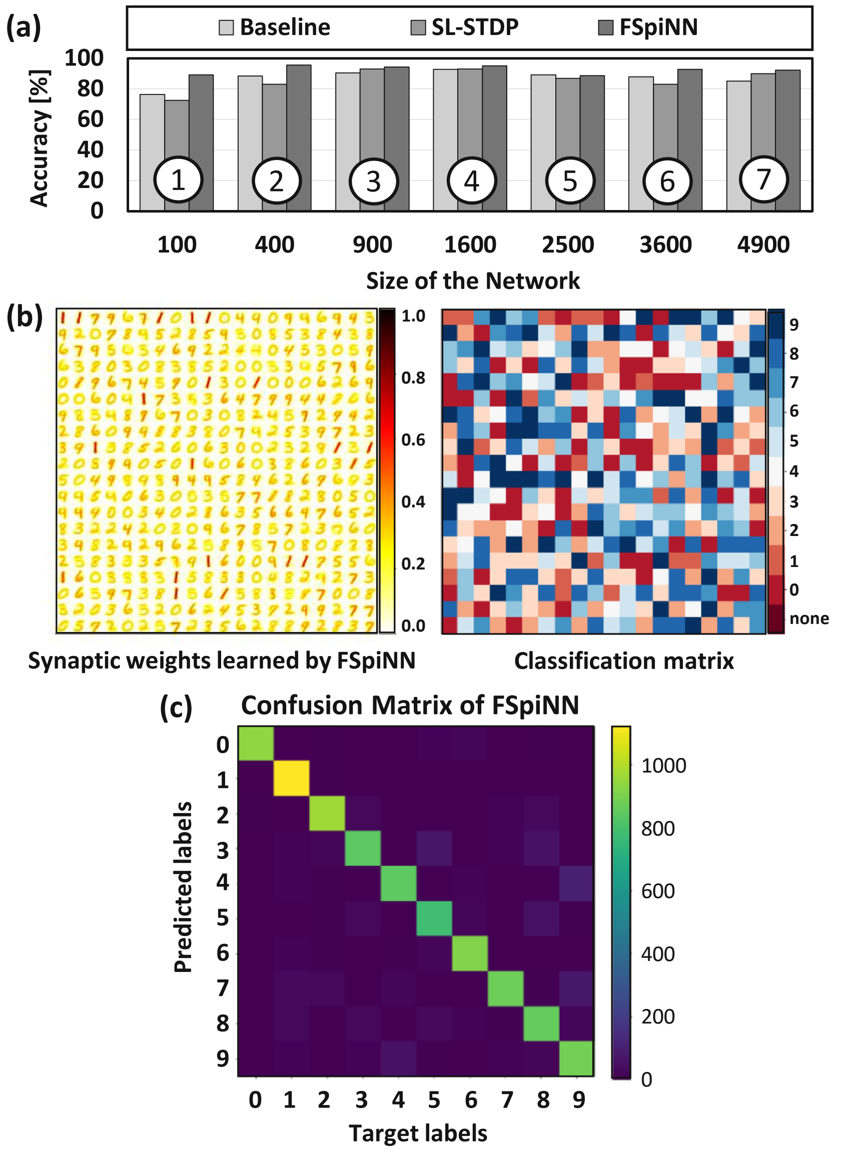


Figure 4.13: (a) Comparisons of accuracy for the MNIST dataset in different sizes of networks: Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900. (b) Synaptic weights learned by FSpiNN and its classification matrix. (c) Confusion matrix in inference phase for Net400.

**Results for the Fashion MNIST Dataset:** Figure 4.14(a) shows the accuracy after 1x epoch of training. It shows that our FSpiNN still maintains (and even improves in certain cases) the accuracy across different sizes of networks as compared to other designs (i.e., baseline and SL-STDP). Following are the detailed accuracy improvements achieved by FSpiNN from the baseline.

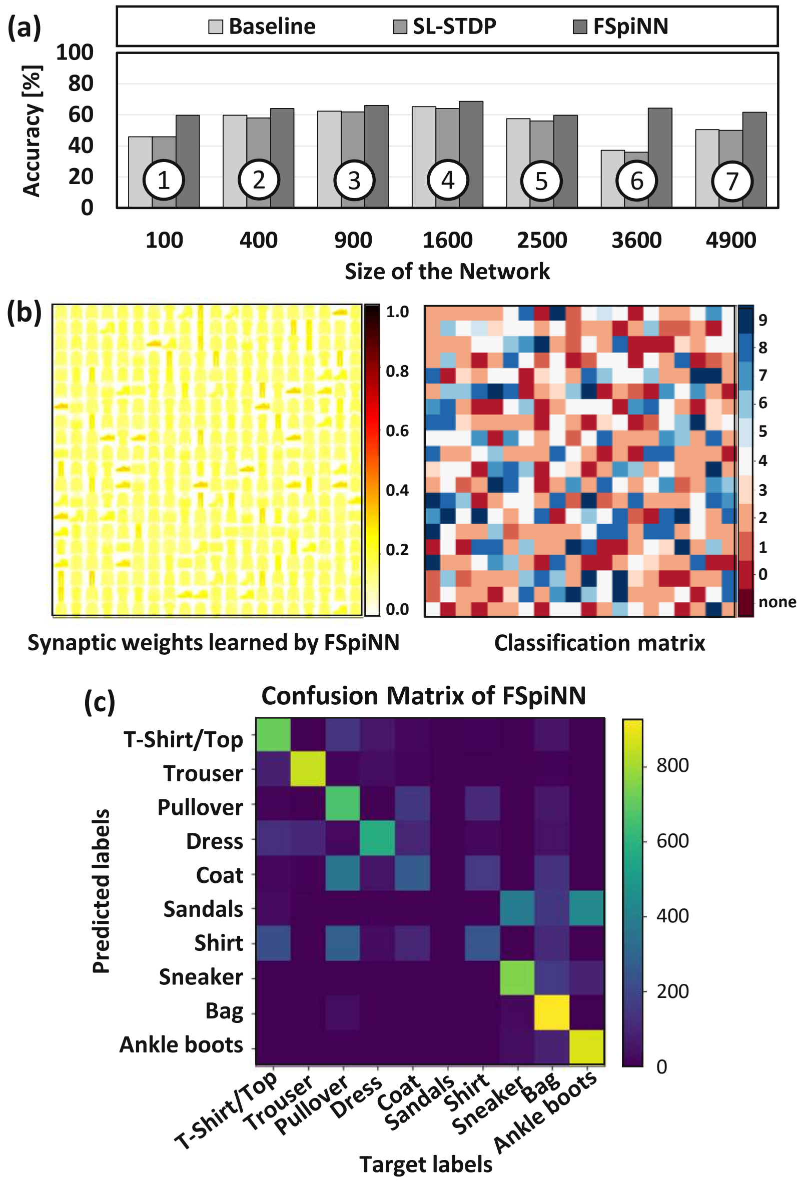


Figure 4.14: (a) Comparisons of accuracy for the Fashion MNIST dataset in different sizes of networks: Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900. (b) Synaptic weights learned by FSpiNN and its classification matrix. (c) Confusion matrix in inference phase for Net400.

- Label-①: In Net100, FSpiNN achieves 14.2% improvement with 60.2% accuracy.
- Label-②: In Net400, FSpiNN achieves 5.2% improvement with 64.8% accuracy.
- Label-③: In Net900, FSpiNN achieves 3.6% improvement with 66% accuracy.
- Label-④: In Net1600, FSpiNN achieves 3.5% improvement with 68.8% accuracy.

- Label-⑤: In Net2500, FSpiNN achieves 3% improvement with 60.6% accuracy.
- Label-⑥: In Net3600, FSpiNN achieves 27% improvement with 64.4% accuracy.
- Label-⑦: In Net4900, FSpiNN achieves 11% improvement with 61.6% accuracy.

Here, we observe the same trend as observed in the MNIST case. A larger network has the potential to achieve higher accuracy because more neurons are available for recognizing more feature variations. This trend is shown in Figure 4.14(a) for Net100-Net1600 and Net3600-Net4900. At the same time, a larger network is harder to train because more synapses have to effectively learn input features. Therefore, a larger network may achieve lower accuracy than the smaller ones in cases where the synapses are not effectively trained. This trend is shown in Figure 4.14(a) for Net1600-Net3600. The reason is that, in our experiments, we keep the same hyper-parameter tuning across different sizes of networks, and only performed 1x epoch of training. Therefore, the accuracy of a larger network could still be improved through more effective hyper-parameter tuning (e.g., more training epochs), as suggested in Figure 4.15. The results in Figure 4.15 indicate that employing multi-epoch training can increase the accuracy, since the same features in the training set are learned multiple times by the network. The accuracy improvement in the earlier epoch is typically higher than in the later ones, thereby only relying on multi-epoch training may incur high energy consumption, without gaining significant accuracy improvement in the end. To address this, our FSpiNN employs the adaptive potentiation factor and inhibition strength, which increase the confidence in learning over time in the training. The results also show that our FSpiNN achieves the highest accuracy across different epochs as compared to state-of-the-art designs. Moreover, FSpiNN with a 1x training epoch achieves higher accuracy than state-of-the-art designs with 3x training epochs. These results show the effectiveness of the learning algorithm in FSpiNN.

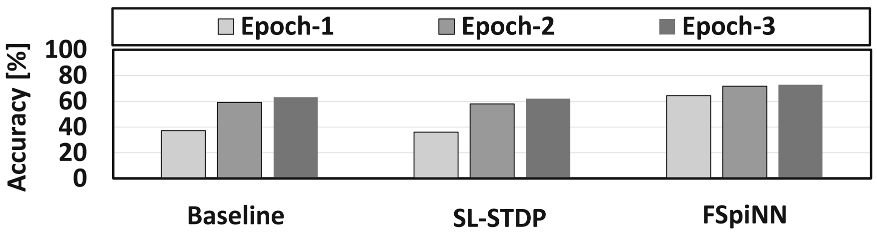


Figure 4.15: Results of accuracy after 3x epochs of training for Net3600 when running the Fashion MNIST.

Furthermore, Figure 4.14(b) shows the synaptic weights and its classification matrix, and Figure 4.14(c) shows the confusion matrix for Net400. These results show the common confusions, such as when identifying between pullover, coat, and shirt, as well as sandals, sneakers, and ankle boots. The reason is that, the connecting synapses from the same neuron learn the common features (shape) from these classes. Hence, the same neuron generates the highest number of spikes for different classes, thereby resulting in more frequent false classifications.

These experimental results also indicate that the input images with more overlapping features are harder to classify. Therefore, in general, the classification accuracy achieved in the MNIST is higher than the Fashion MNIST, since the MNIST has relatively simpler features than the Fashion MNIST. However, our FSpiNN can still achieve better accuracy in the Fashion MNIST across different sizes of networks, outperforming state-of-the-art designs. The maintained accuracy achieved by our FSpiNN comes from the improved STDP-based learning, which reduces the spurious weight updates, and employs an effective STDP potentiation and inhibition strength.

#### 4.2.6.2 Impact of the Fixed-Point Quantization on Accuracy

Our framework converts a floating-point (FP32) format to a fixed-point format, and conducts exploration to study the impact of different quantization levels on the accuracy.

**Results for the MNIST Dataset:** Figure 4.16 shows the experimental results for the MNIST. Label-① shows that the FSpiNN achieves better accuracy than the baseline and the SL-STDP, when the minimum bit-width of quantization is 8 bits. The reason is that, the 8-bit (or more) format in the FSpiNN provides sufficient levels of weight values to modulate the input spikes from the MNIST images, and induce each neuron to recognize a specific digit class. In 8-bit precision, our FSpiNN achieves 91.6% accuracy, while the baseline and the SL-STDP achieve 87.6% and 82%, respectively. It indicates that the accuracy achieved by the FSpiNN 8-bit is slightly less than the FSpiNN FP32 (pointed by label-②), but still higher than the baseline and the SL-STDP with FP32 precision (pointed by label-③ and label-④, respectively). Therefore, the FSpiNN 8-bit offers no accuracy loss with a reduced bitwidth for the MNIST.

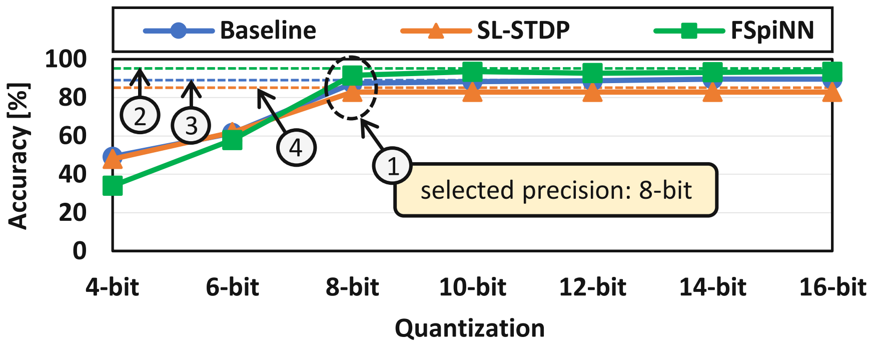


Figure 4.16: Accuracy vs. quantization for the MNIST dataset for Net400.

**Results for the Fashion MNIST Dataset:** Figure 4.17 shows the experimental results for the Fashion MNIST. Label-① shows that the FSpiNN achieves better accuracy than the baseline and the SL-STDP, when the minimum bitwidth of quantization is 8 bits. The reason is that, the 8-bit (or more) format in the FSpiNN provides sufficient levels of weight values to modulate the input spikes from the Fashion MNIST images, and induce each neuron to recognize a specific fashion class. In 8-bit precision, our FSpiNN achieves 64.8% of accuracy, while the baseline and the SL-STDP achieve 59.2% and 58%,

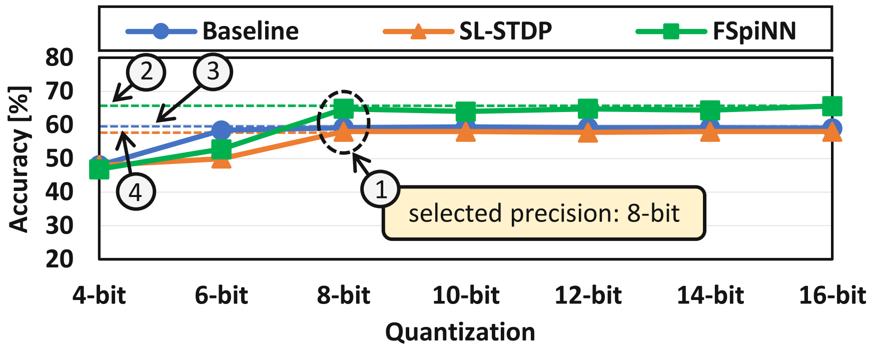


Figure 4.17: Accuracy vs. quantization for the Fashion MNIST dataset for Net400.

respectively. It indicates that the accuracy achieved by the FSpiNN 8-bit is comparable to the FSpiNN FP32 (pointed by label-②), and it is higher than the baseline and the SL-STDP with FP32 precision (pointed by label-③ and label-④, respectively). Therefore, the FSpiNN 8-bit offers no accuracy loss with a reduced bitwidth for the Fashion MNIST.

These experimental results also show that, for both the MNIST and Fashion MNIST datasets, the quantization levels with less than 8-bit precision do not provide sufficient unique information for distinguishing features of different classes in the input images. This condition reduces the efficacy of STDP learning of the synapses and recognition capability of the neurons, thereby leading to low classification accuracy. Furthermore, a reduced bitwidth is beneficial since it leads to a reduced memory requirement and energy consumption, which will be discussed in Section 4.2.6.3 and Section 4.2.6.4. Note, the users can select the quantization level based on the trade-off consideration in the design specifications (e.g., accuracy, memory, and power/energy budget).

#### 4.2.6.3 Reducing the Memory Requirements

Figure 4.18 shows the memory requirements of different designs across different sizes of networks for both the training and inference phases. Label-① shows that, Net3600 and Net4900 that employ the baseline or the SL-STDP techniques consume more than 100 MB, thereby making them difficult to be deployed on embedded systems. Meanwhile, our FSpiNN without quantization (FP32) achieves 1.8x and 1.9x memory savings as compared to the baseline, for Net3600 and Net4900, respectively. The reason is that, the FSpiNN FP32 removes the inhibitory neurons completely, thereby preventing their parameters to be saved in the memory. After applying quantization, the memory requirement is reduced even more. The FSpiNN 16-bit achieves about 3.6x and 3.7x memory savings, while the FSpiNN 8-bit achieves about 7.3x and 7.5x memory savings, when compared to the baseline for Net3600 and Net4900, respectively. Figure 4.18 also shows that the FSpiNN 8-bit consumes about 0.16 MB - 28 MB for Net100-Net4900, thereby making the networks easier to be deployed on embedded systems. Furthermore, if we consider the accuracy that the quantized designs can achieve, we can select the FSpiNN design that offers a good trade-off between high accuracy and an acceptable memory footprint.

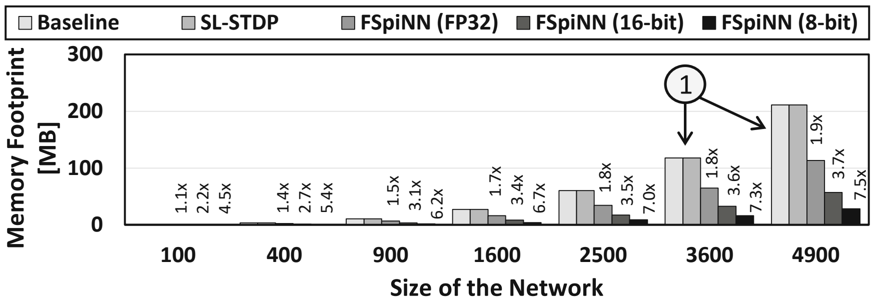


Figure 4.18: Memory requirements for different sizes of networks (i.e., Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900) and different quantization levels (i.e., FP32/without quantization, 16-bit, and 8-bit).

### 4.2.6.4 Energy Efficiency Improvements

Figure 4.19 and Figure 4.20 illustrate the energy efficiency across different sizes of networks and different GPUs for the MNIST and Fashion MNIST datasets, respectively. These figures show that the SL-STDP achieves higher energy efficiency than the baseline in the training phase, and our FSpiNN achieves the highest energy efficiency among all designs in both the training and inference phases.

**Training:** The SL-STDP improves the energy efficiency by 1.1x-1.2x as compared to the baseline, across different sizes of networks and GPUs, for both the MNIST and Fashion MNIST datasets. The reason is that, the SL-STDP only employs postsynaptic spike-based weight updates whose number of updates is less than the baseline, which employs presynaptic and postsynaptic spike-based weight updates. The FSpiNN FP32 improves the energy efficiency more than the SL-STDP, that is by 1.1x-2.8x (MNIST) and by 1.1x-1.9x (Fashion MNIST) as compared to the baseline. The reason is that, apart from the elimination of presynaptic spike-based weight updates, the FSpiNN FP32 also eliminates the inhibitory neurons and reduces the STDP complexity. After applying quantization, the FSpiNN 16-bit and the FSpiNN 8-bit improve the energy efficiency even more than the FSpiNN FP32. That is, the FSpiNN 16-bit achieves 1.7x-3.9x (MNIST) and 1.2x-2.4x (Fashion MNIST), while the FSpiNN 8-bit achieves a 1.8x-4.3x (MNIST) and 1.5x-2.7x (Fashion MNIST), as compared to the baseline.

**Inference:** The SL-STDP has comparable energy efficiency as compared to the baseline, across different sizes of networks and GPUs, for both the MNIST and Fashion MNIST datasets. The reason is that, the SL-STDP and the baseline have similar computational complexity in the inference phase. Meanwhile, the FSpiNN FP32 improves the energy efficiency by 1.3x-1.9x (MNIST) and by 1.1x-1.4x (Fashion MNIST) as compared to the baseline. The improvements mainly come from the elimination of the inhibitory neurons. After applying quantization, the FSpiNN 16-bit and the FSpiNN 8-bit improve the energy efficiency even more than the FSpiNN FP32. That is, the FSpiNN 16-bit achieves 1.4x-2.6x (MNIST) and 1.2x-2.1x (Fashion MNIST), while the FSpiNN 8-bit achieves 1.4x-2.9x (MNIST) and 1.3x-2.3x (Fashion MNIST), compared to the baseline.

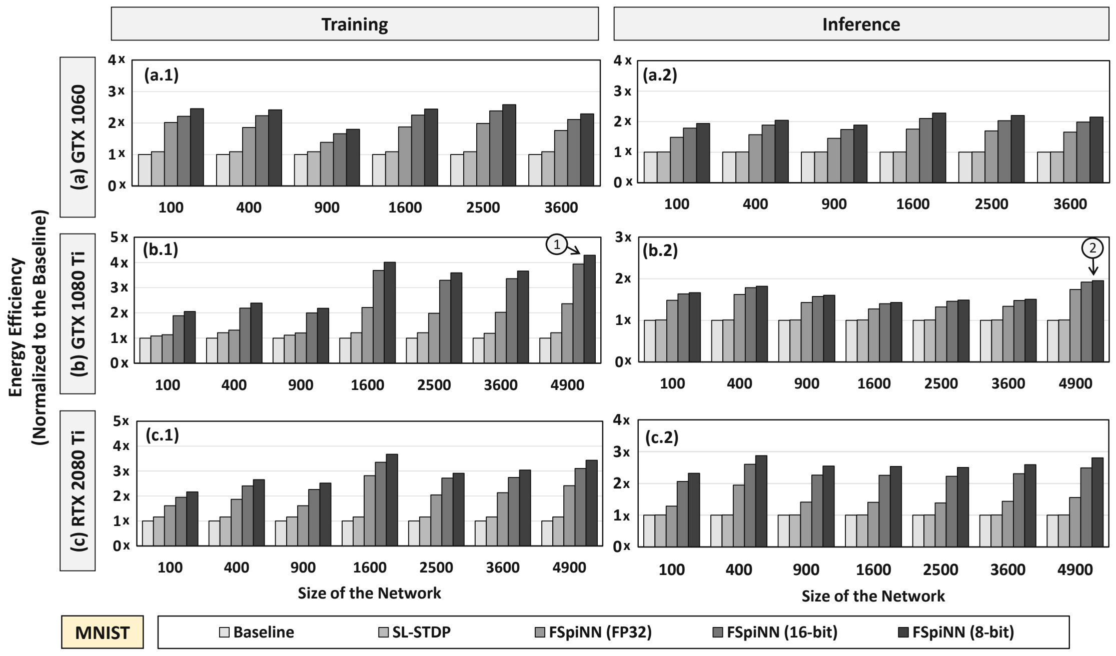


Figure 4.19: Results of energy efficiency (normalized to the baseline) for training and inference with the MNIST datset, while considering different sizes of networks (i.e., Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900), different quantization levels (i.e., FP32, 16-bit, and 8-bit), and different types of GPUs: Nvidia GeForce (a) GTX 1060, (b) GTX 1080 Ti, and (c) RTX 2080 Ti. Due to the limited memory, the GTX 1060 can only run Net100-Net3600.

Furthermore, if we consider the classification accuracy and memory footprint that the quantized designs can achieve, we can select the FSpiNN design that offers a good trade-off in accuracy, memory, and energy efficiency. For instance, the FSpiNN 8-bit achieves energy-efficiency improvements by 4.3x (MNIST) and by 2.7x (Fashion MNIST) in training (see label-① in Figure 4.19 and Figure 4.20), and by 2x (MNIST) and by 1.6x (Fashion MNIST) in inference (see label-② in Figure 4.19 and Figure 4.20), as compared to the baseline in Net4900, while obtaining 7.5x memory saving with accuracy of ∼92% for the MNIST and ∼61% for the Fashion MNIST. The experimental results in Figure 4.19 and Figure 4.20 also suggest that our FSpiNN framework is scalable for different sizes of networks, and can be used for other systems where different types of GPUs are deployed, such as embedded systems with embedded GPUs.

Note that this work is not about justifying SNNs over deep neural networks (DNNs). Rather, we consider what necessary optimizations are required if the SNNs make it to a real-world system following an increasing trend of neuromorphic computing, due to their benefits in energy-efficient spike-based computations and unsupervised learning. Moreover, there is a substantial difference in the underlying learning mechanism between the SNNs (with unsupervised learning) and the DNNs (with supervised learning), thus

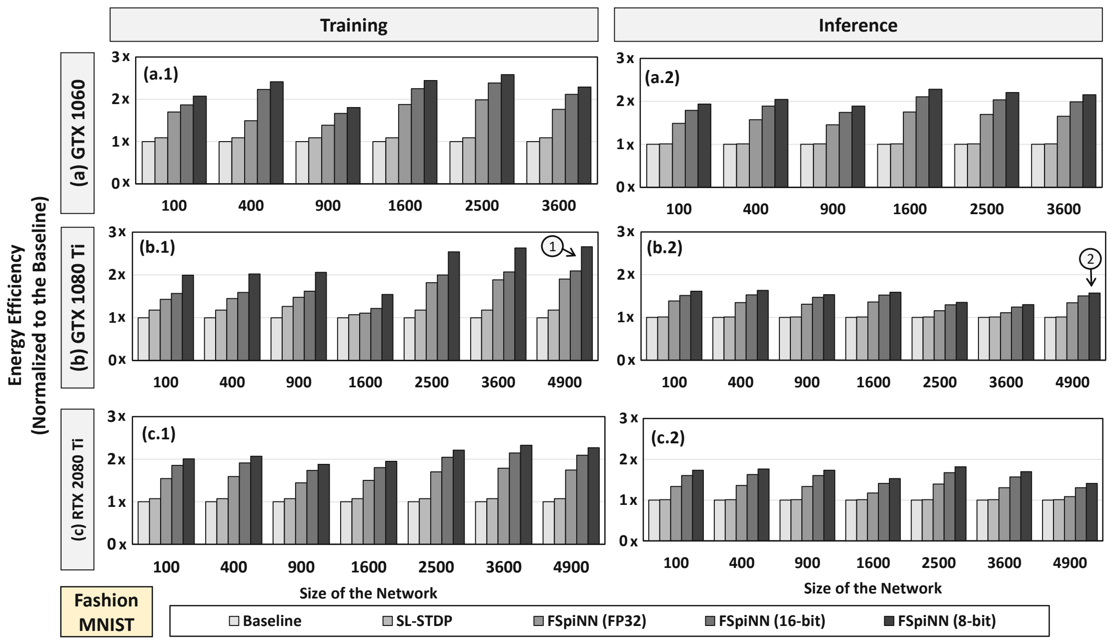


Figure 4.20: Results of energy-efficiency (normalized to the baseline) for training and inference with the Fashion MNIST dataset, while considering different sizes of networks (i.e., Net100, Net400, Net900, Net1600, Net2500, Net3600, and Net4900), different quantization levels (i.e., FP32, 16-bit, and 8-bit), and different types of GPUs: Nvidia GeForce (a) GTX 1060, (b) GTX 1080 Ti, and (c) RTX 2080 Ti. Due to the limited memory, the GTX 1060 can only run Net100-Net3600.

we cannot directly compare the accuracy of the unsupervised SNNs with the supervised DNNs. Previous work [DBDRC+15] has observed that the accuracy of the DNNs (with the supervised back-propagation algorithm) is generally higher than the SNNs (with the unsupervised STDP algorithm), because the unsupervised STDP algorithm does not have labels when updating the weights, hence it is less effective than the supervised ones. Furthermore, in the SNN community, many different optimization aspects are explored, and they have the potential to be incorporated into our FSpiNN framework. For instance, the works in [MMS+11] and [ROD+10] focus on generating precise spike sequences like real-world observation. They target a different optimization purpose compared to the one targeted by our FSpiNN framework. However, they can still be incorporated into the FSpiNNs' optimization flow for generating precise spike sequences. This illustrates the flexibility of our FSpiNN for integration with other optimization techniques.

### 4.2.6.5 Further Discussion

The proposed compression techniques for SNNs have several clear differences over the deep compression work [HMD16] which is designed for DNNs, as highlighted the following.

- Edge (connection) removal in the proposed compression techniques replaces the inhibition spikes which come from the inhibitory neurons with the excitatory spikes from the excitatory neurons, hence it does not depend on the weights of inhibitory connections. In this manner, all inhibitory neurons and synapses are completely removed from the network. Meanwhile, edge removal in the deep compression work eliminates all connections with weights below a pre-defined threshold value from the network.
- Optimization in the proposed compression techniques focuses on the SNN-specific operations, which do not exist in the deep compression work. For instance, the complexity of the bio-inspired STDP learning rule and the number of weight updates during STDP learning process are reduced.
- To compensate the loss of accuracy induced by the optimization process, the proposed compression techniques employ adaptive potentiation factor, timestep-based weight updates, and inhibition strength adjustments, while the deep compression work mainly relies on the retraining approach.

### 4.2.7 Summary of FSpiNN Framework

The proposed FSpiNN framework synergistically employs different optimization techniques to reduce the memory footprint and improve the energy efficiency of SNNs, while maintaining their accuracy. Experimental results illustrate the benefits and efficiency of the proposed framework as compared to the state-of-the-art designs, across different sizes of networks and different datasets (MNIST and Fashion MNIST). For instance, in a network with 4900 excitatory neurons, our FSpiNN achieves 7.5x memory saving, as well as 3.5x energy-efficiency improvement on average for training and 1.8x on average for inference, with no accuracy loss. In short, our proposed framework enables efficient embedded SNN implementations for the next-generation smart embedded systems.

## 4.3 Q-SpiNN: A Framework for Quantizing SNNs

This section aims at addressing **Problem-2**, i.e., the solution for systematically employing quantization on SNNs to maximize memory saving, while maintaining accuracy.

### 4.3.1 Motivational Study

We observe that, apart from the weights, there are other SNN parameters that can be quantized to further reduce the memory footprint as they also need to be stored in the on-chip memory [SVR17, RVG$^{+}$17], e.g., neurons' membrane and threshold potentials. To see the potential of such an idea, we study the impact of different *precision levels* (bitwidth) for different SNN parameters on the accuracy through experiments. Here, we consider the FC-based network architecture in Figure 4.8(a) and run it through PyTorch-based simulation on GPGPU, i.e., Nvidia RTX 2080 Ti. The detailed experimental setup is explained in Section 4.3.5. Figure 4.21 shows the experimental results, from which we make the following key observations.

- Different SNN parameters may need different integer bitwidth representation, as they have a different range of values.
- Different combinations of precision levels (bitwidth) may achieve comparable accuracy, but occupy different memory footprints. For instance, W($Q$1.16)-N(FP32), W(FP32)-N($Q$11.16), and W($Q$1.16)-N($Q$11.16) obtain about 84% accuracy, while consuming about 1.2 MB, 0.68 MB, 1.19 MB, and 0.67 MB respectively, as indicated by label-❶ in Figure 4.21.
- Less memory footprint requires a less number of memory accesses, and thereby less access energy. This potentially improves the energy efficiency of SNN processing, as the memory accesses dominate the energy of SNN processing (i.e., 50%-75% of total system energy) [KSVR19].

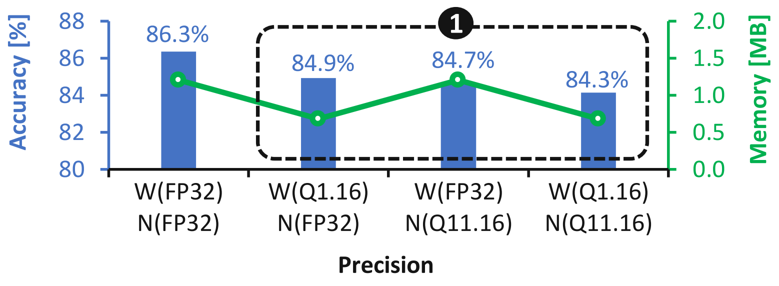


Figure 4.21: Accuracy and memory footprint of an SNN with 400 excitatory neurons with different precision levels on the MNIST dataset [LBBH98a]. The W($X$)-N($Y$) denotes $X$ precision format for the weights and $Y$ precision format for the neuron parameters (i.e., neurons' membrane and threshold potentials).

## 4.3.2 Scientific Research Challenges

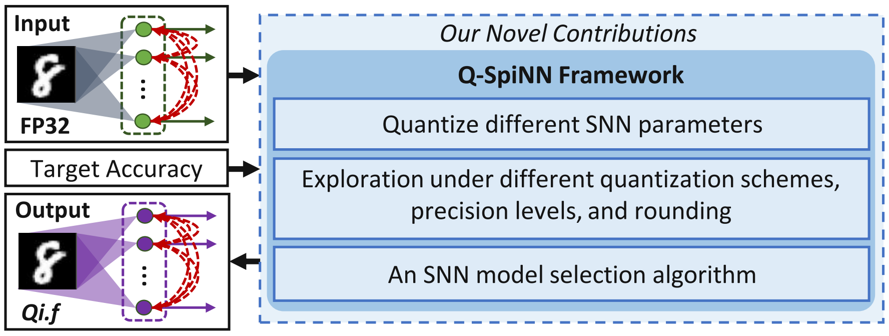
Although the quantization effectively reduces the memory footprint, it leads to accuracy degradation if the quantization process is not performed carefully. Furthermore, finding the appropriate quantization levels for different SNN parameters is challenging, as the number of potential combinations of precision levels is large. Therefore, the key challenge is *how to effectively perform quantization and exploit the trade-off between memory and accuracy, so that the memory footprint is reduced and the targeted accuracy is met.*

***Required:*** *A systematic quantization approach for SNN parameters (i.e., weights and neuron parameters) to maximize the memory saving while maintaining accuracy, thereby enabling the SNN deployment on many AI applications under tight memory budgets.*

## 4.3.3 Novel Contributions

To address the above challenges, we propose *Q-SpiNN, a novel Quantization framework for Spiking Neural Networks* [PS21a], through the following mechanisms (the overview is shown in Figure 4.22).

- **Employing quantization for different SNN parameters** based on their significance to the accuracy, which are analyzed by observing the accuracy obtained under different precision levels.
- **Exploring different combinations of quantization schemes, precision levels, and rounding schemes** to find the SNN models that meet the user-targeted accuracy, and refer them to as the solution candidates.
- **Developing an algorithm to select the SNN model from the given candidates**. It quantifies the benefit of the memory-accuracy trade-off obtained by the candidates using the proposed reward function, and then selects the one with the highest benefit.



Input
FP32
Target Accuracy
Output
Qi.f
Our Novel Contributions
Q-SpiNN Framework
Quantize different SNN parameters
Exploration under different quantization schemes, precision levels, and rounding
An SNN model selection algorithm

Figure 4.22: Overview of our novel contributions (shown in the blue boxes).

### 4.3.4 Q-SpiNN Framework

The Q-SpiNN framework employs the following key steps for obtaining memory-efficient SNNs while meeting the target accuracy (the overview is in Figure 4.23).

1. **Quantization of different parameters (Section 4.3.4.1)** through the following.
   - Maximizing the quantization for each SNN parameter.
   - Defining the precision level (bitwidth) for each parameter based on its significance, that is obtained by analyzing the accuracy under different precision levels.
2. **Exploration of different quantization approaches (Section 4.3.4.2)** through the following means.
   - Observing the accuracy obtained from different quantization schemes (PTQ and ITQ), different precision levels, and different rounding schemes (TR, RN, and SR).
   - Selecting the SNN models that meet the target accuracy as the solution candidates.
3. **An SNN model selection (Section 4.3.4.3)** that finds an appropriate SNN model from the given candidates through the following means.
   - Quantifying the benefit of the memory-accuracy trade-off obtained by the SNN model candidates using *our proposed multi-objective reward function*,
   - Selecting the SNN model with the highest benefit.

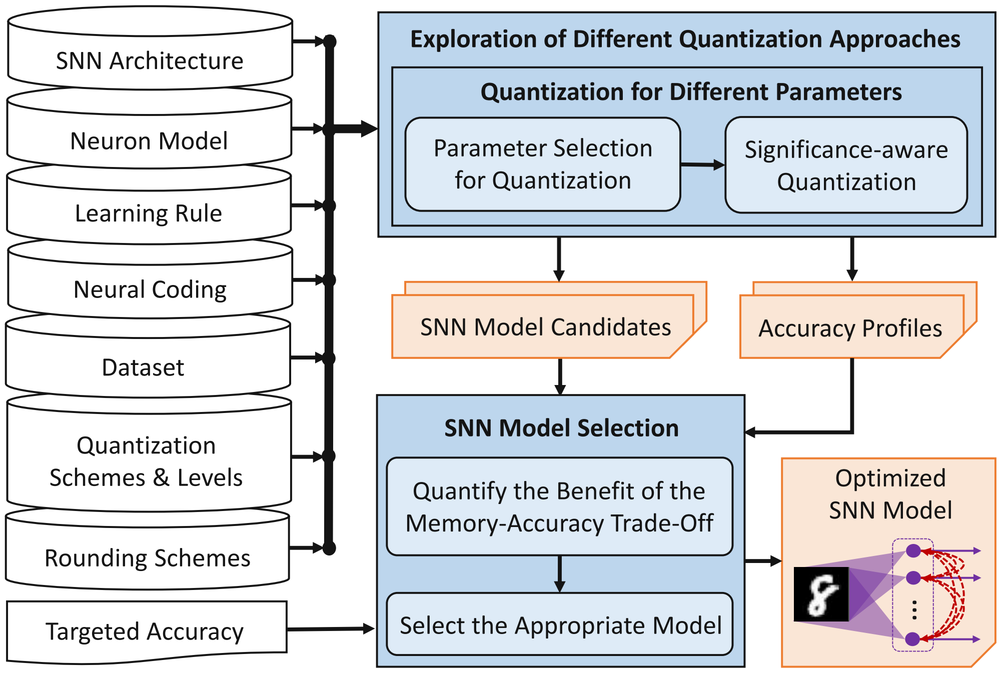

SNN Architecture
Neuron Model
Learning Rule
Neural Coding
Dataset
Quantization Schemes & Levels
Rounding Schemes
Targeted Accuracy
Exploration of Different Quantization Approaches
Quantization for Different Parameters
Parameter Selection for Quantization
Significance-aware Quantization
SNN Model Candidates
Accuracy Profiles
SNN Model Selection
Quantify the Benefit of the Memory-Accuracy Trade-Off
Select the Appropriate Model
Optimized SNN Model

Figure 4.23: An overview of the Q-SpiNN framework. The novel contributions are highlighted in blue.

#### 4.3.4.1 Quantization of Different SNN Parameters

Different SNN designs may have different SNN parameters that can be quantized. Therefore, *to provide a generic solution for any SNN designs, we propose significance-aware quantization steps* (the overview is shown in Figure 4.24). The idea is to maximize the quantization for each SNN parameter, and define the precision level for each parameter based on its significance to the accuracy. For the given SNN model (in FP32), we first determine the parameters to be quantized by manually selecting them. Afterward, we analyze the significance of each parameter to determine the integer and fractional bitwidth. For the integer part, the bitwidth requirement is analyzed by observing the range of parameter values when running the given workload. For the fractional part, there are two cases. If the parameter is a constant, then the bitwidth depends on the parameter value; and otherwise (if the parameter is a variable), the bitwidth requirement is analyzed by gradually reducing its precision and observing the output accuracy. In this manner, the impact of the parameters' bitwidth on accuracy is systematically explored.

**Case Study:** We provide a case study to show how the proposed quantization steps are done for the unsupervised SNN with the MNIST dataset; see the reference for the unsupervised SNN in Section 2.3 and Figure 2.12(a). First, we select the $w$, $v$, $v_{th}$, $v_{reset}$, and $\theta$ as the parameters to be quantized. $v_{reset}$ and $\theta$ are *constants*, while others (i.e., $w$, $v$, and $v_{th}$) are *variables* in the training.

- *For constant parameters:* We quantize the constant parameters based on their values

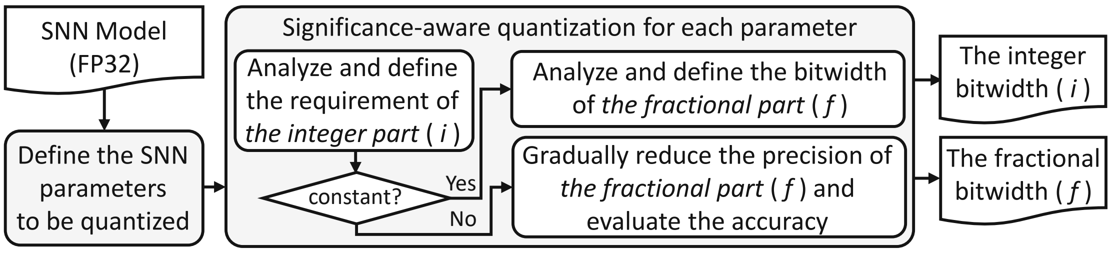

SNN Model (FP32)
Define the SNN parameters to be quantized
Significance-aware quantization for each parameter
Analyze and define the requirement of the integer part ( i )
constant?
Yes
No
Analyze and define the bitwidth of the fractional part ( f )
Gradually reduce the precision of the fractional part ( f ) and evaluate the accuracy
The integer bitwidth ( i )
The fractional bitwidth ( f )

Figure 4.24: Overview of the proposed quantization steps for the given SNN model.

(see Table 4.2). For instance, $v_{reset}$ = -65 mV, and it is represented using 8 bits with $Q7.0$ format, while $\theta = 0.05$ mV and it is represented using 8 bits with $Q1.6$ format. In this manner, 24 bits are saved for $v_{reset}$ and $\theta$, as compared to the FP32.

- *For variable parameters:* We perform experiments to obtain the ranges of parameter values (see the values in Table 4.2).
  - For the integer part, we define the integer bitwidth based on the observed ranges, i.e., $v_{th}$, $v$, and $w$ need 11 bits, 11 bits, and 1 bit of integer, respectively.
  - For the fractional part, we gradually reduce the precision and observe the output accuracy to study the impact of different precision levels. Therefore, *we perform a design exploration*, which is discussed further in Section 4.3.4.2.

Table 4.2: Observed unsupervised SNN parameter values for the MNIST workload.

| Parameters | Value | Description |
|---|---|---|
| $v_{reset}$ | -65 mV | shown by label-① in Figure 4.25(a) |
| $\theta$ | 0.05 mV | shown by label-② in Figure 4.25(a) |
| $v_{th}$ | -52 mV – 1271.88 mV | shown by label-③ in Figure 4.25(a) |
| $v$ | -887.29 mV – 1250.18 mV | shown by label-④ in Figure 4.25(a) |
| $w$ | 0 – 0.7 | shown by label-⑤ in Figure 4.25(b) |

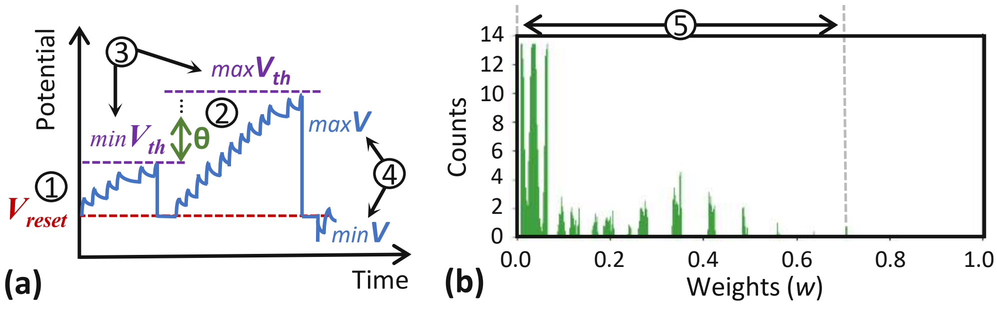


Figure 4.25: (a) Overview how neuron parameters are involved in neuronal dynamics. (b) The weight distribution of the trained unsupervised SNN with 400 excitatory neurons.

### 4.3.4.2 Exploration of Different Quantization Approaches

To find the effective configuration of quantization for the given SNN, a design exploration of different quantization approaches is required. Therefore, *we comprehensively study the impact of different quantization schemes (PTQ and ITQ), different precision levels, and different rounding schemes (TR, RN, and SR), and select the ones that meet the targeted accuracy.* To do this, we devise an algorithm to systematically perform exploration across a range of selected parameters (pseudo-code is in Algorithm 6). This algorithm employs the following steps (considering an example for the unsupervised SNN case).

- We train the given model with a floating-point precision (Algorithm 6: line 3), and the test accuracy of the trained model is considered as the baseline accuracy (Algorithm 6: line 4). Then, we perform the PTQ and the ITQ schemes, subsequently.
- For the PTQ, the quantization is performed on the trained model, then the accuracy is evaluated (Algorithm 6: lines 12-16). Meanwhile, for the ITQ, we quantize the given SNN model during the training. Therefore, the trained model is already in a quantized form, and can be used for the accuracy evaluation (Algorithm 6: lines 18-23).
- For both schemes, we reduce the precision of each parameter using a nested for-loop (Algorithm 6: lines 7-9), and in each step, we explore the use of different rounding schemes (Algorithm 6: line 10). The depth of the loop depends on the parameters (e.g., we consider $w$, $v$, and $v_{th}$ for the unsupervised SNN case). If the accuracy is within the target, then the model is selected as a solution candidate. Otherwise, the currently investigated precision and the lower precision (if any) for the corresponding parameter, are not considered in the next exploration steps (Algorithm 6: lines 24-33). *Therefore, the design space is reduced and the exploration is performed efficiently.*

This exploration populates the SNN models that meet the target accuracy (i.e., the solution candidates). To select the appropriate model out of them, we propose a model selection algorithm, which is discussed in Section 4.3.4.3.

### 4.3.4.3 SNN Model Selection

We obtain a set of model candidates (with their corresponding accuracy and quantization) from the exploration in Section 4.3.4.2. Afterward, we need to select the Pareto-optimal model out of the candidates, while considering the accuracy and the memory footprint. Towards this, *we propose an SNN model selection algorithm that quantifies the benefit of the memory-accuracy trade-off obtained by the candidates using the proposed multi-objective reward function.* The idea of our reward function is to prioritize the model with higher accuracy and a smaller memory footprint, which is expressed as Equation 4.4.

$$R(acc_q, m_{norm}) = acc_q - \mu\, m_{norm} \tag{4.4}$$

$$m_{norm} = \frac{mem_q}{mem_{fp}} \tag{4.5}$$

**Algorithm 6** Pseudo-code of the proposed exploration

```
INPUT: (1) SNN: floating-point model ($m_{fp}$), accuracy ($acc_{fp}$), parameters ($m_{fp}.w, m_{fp}.v$, $m_{fp}.v_{th}$);
    (2) Maximum allowed accuracy degradation ($acc_{deg}$); (3) Quantization schemes ($QS = [PTQ, ITQ]$);
    (4) Rounding schemes ($RS = [TR, RN, SR]$); (5) Precision levels ($QL$); // $QL$ is the user-defined
    fractional bitwidth sorted in descending order, e.g., $QL = [16, 14, ..., 0]$;
OUTPUT: SNN model candidates ($C$);
BEGIN
    Initialization:
1:  $C = []$;
2:  $c = 1$;
    Process:
3:  $\overline{m}_{fp} \leftarrow train(m_{fp})$;
4:  $acc_{fp} = test(\overline{m}_{fp})$;
5:  for ($qs = 1$ to $len(QS)$) do
6:      $Nw = len(QL)$; $Nv = len(QL)$; $Nt = len(QL)$;
7:      for ($iw = 1$ to $Nw$) do
8:          for ($iv = 1$ to $Nv$) do
9:              for ($it = 1$ to $Nt$) do
10:                 for ($rs = 1$ to $len(RS)$) do
11:                     if ($QS[qs] == PTQ$) then
12:                         $w_q = quantize(\overline{m}_{fp}.w, QL[iw], RS[rs])$;
13:                         $v_q = quantize(\overline{m}_{fp}.v, QL[iv], RS[rs])$;
14:                         $v_{thq} = quantize(\overline{m}_{fp}.v_{th}, QL[it], RS[rs])$;
15:                         $\overline{m}_q \leftarrow substitute(\overline{m}_{fp}, (w_q, v_q, v_{thq}))$;
16:                         $acc_q = test(\overline{m}_q)$;
17:                     else
18:                         $w_q = quantize(m_{fp}.w, QL[iw], RS[rs])$;
19:                         $v_q = quantize(m_{fp}.v, QL[iv], RS[rs])$;
20:                         $v_{thq} = quantize(m_{fp}.v_{th}, QL[it], RS[rs])$;
21:                         $m_q \leftarrow substitute(m_{fp}, (w_q, v_q, v_{thq}))$;
22:                         $\overline{m}_q \leftarrow train(m_q)$;
23:                         $acc_q = test(\overline{m}_q)$;
24:                     if ($acc_q \geq (acc_{fp} - acc_{deg})$) then
25:                         $C[c] = \overline{m}_q$;
26:                         $c$ += 1;
27:                     else
28:                         if ($iw \geq 1$)&($iv == 1$)&($it == 1$) then
29:                             $Nw = iw - 1$;
30:                         else if ($iw == 1$)&($iv \geq 1$)&($it == 1$) then
31:                             $Nv = iv - 1$;
32:                         else if ($iw == 1$)&($iv == 1$)&($it \geq 1$) then
33:                             $Nt = it - 1$;
34: return $C$;
END
```

$$mem = mem\_w + mem\_n = Nw\ Bw + \sum_{k} Nn^k\ Bn^k \tag{4.6}$$

$acc_q$ denotes the test accuracy of the quantized SNN model, $m_{norm}$ denotes the normalized memory footprint, and the coefficient $\mu$ is the weight to trade-off between memory and accuracy. Note $acc$, $m_{norm}$, and $\mu$ have a value range of [0,1]. $m_{norm}$ is obtained from the ratio between the memory of the quantized model ($mem_q$) and the floating-point model

($mem_{fp}$), as stated in Equation 4.5. The memory footprint ($mem$) is estimated by the total memory required by the weights ($mem\_w$) and neuron parameters ($mem\_n$), as shown in Equation 4.6. $mem\_w$ is obtained by multiplying the number of weights ($Nw$) and the respective bitwidth ($Bw$). A similar approach is used for neuron parameters, i.e., multiplying the number of parameters ($Nn$) and the bitwidth ($Bn$). Since the neuron has several parameters ($k$) which may have different precision, $mem\_n$ is defined as the total bits from all neuron parameters.

## 4.3.5 Evaluation Methodology

Figure 4.26 shows the experimental setup for evaluating the Q-SpiNN framework. We use the PyTorch-based simulation to evaluate the accuracy of the unsupervised SNN [HSK+18] and the supervised SNN [KMN20], estimate the memory, and select the SNN model. We run the simulations on GPGPU machine (i.e., Nvidia GeForce RTX 2080 Ti [Nvic]) and Embedded GPU machine (i.e., Nvidia Jetson Nano [Nvid]) to show the applicability of the Q-SpiNN framework on different hardware platforms with different compute and memory capabilities.

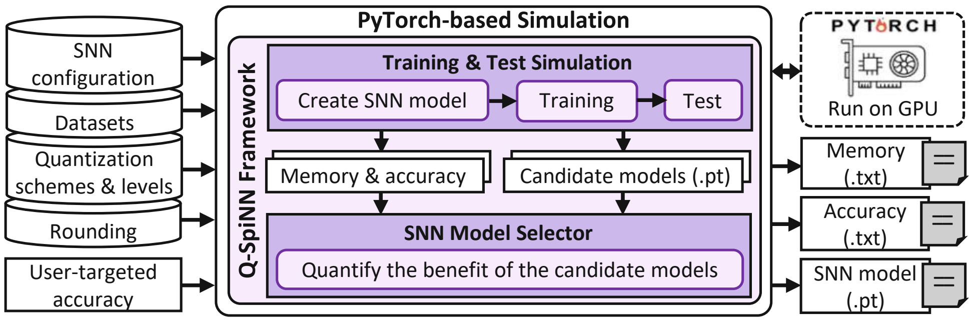


Figure 4.26: Experimental setup for evaluating our Q-SpiNN framework.

**Networks:** We use networks with different architectures, number of layers, and learning rules to show the generality of our Q-SpiNN. For the unsupervised SNN (i.e., U-SNN), we consider an FC-based network with the STDP, as shown in Figure 4.8(a), while for the supervised SNN (i.e., S-SNN), we consider a multi-layer convolutional network with the DECOLLE, as shown in Figure 2.12(b).

**Datasets:** We use the MNIST dataset [LBBH98a] for the U-SNN case, and the DVS-Gesture dataset [ATB+17] for the S-SNN case. In the MNIST, there are 60,000 images for the training and 10,000 images for the test, each having a dimension of 28x28 pixels. Meanwhile, the DVS-Gesture, which is obtained using a Dynamic Vision Sensor (DVS), has 1,342 instances of a set of 11 hand and arm gestures. They are collected from 29 subjects under 3 lighting conditions. Gestures from 23 subjects are used as the training set, and the remaining 6 subjects are used as the test set. Each gesture consists of a stream of events and lasts for 6 seconds. The event streams are downsized from 128x128 to 32x32 by summing the events from 4 neighboring pixels as a common stream [KMN20].

**Comparisons:** We use networks with different precision levels as the comparison partners, for both the U-SNN and S-SNN cases. For the U-SNN, we consider a network from Figure 4.8(a) with 400 excitatory neurons with 1 training epoch (i.e., using the STDP during forward propagation). For the S-SNN, we train the network using the DECOLLE network [KMN20] from Figure 2.12(b) with 200 epochs. For both cases, the baseline refers to the network with FP32 precision for all parameters.

**Quantization Format:** We use the W($X$)-N($Y$) format to represent a model with $X$ precision for the weights and $Y$ precision for the neuron parameters (see Section 2.5). For conciseness, we simply use W($Qi.f$) to represent a model with W($Qi.f$)-N(FP32) precision, and N($Qi.f$) to represent a model with W(FP32)-N($Qi.f$) precision. Furthermore, since there are several neuron parameters involved in the quantization process, their integer part is simply written as $i$, e.g., N($Qi.8$) means that each neuron parameter employs integer bitwidth based on its value range and 8-bit fraction.

## 4.3.6 Results and Discussion

### 4.3.6.1 Impact of Different Quantization Approaches on Accuracy

**Accuracy of the Unsupervised SNN:** In the U-SNN case, we quantize the weights ($w$) and the neuron parameters ($v_{reset}$, $\theta$, $v$, and $v_{th}$), and the experimental results are shown in Figure 4.27. Here, N($Qi.f$) represents the precision of variables $v$ and $v_{th}$. Notable accuracy degradation from the baseline accuracy is observed when the weights' bitwidth is reduced to the 4-bit fraction, as pointed out by label-① for the PTQ and label-② for the ITQ. The reason is that the 4-bit fraction (or fewer) for weights does not have sufficient levels of value to modulate the input spikes, thereby making the learning process ineffective. Meanwhile, quantizing $v$ and $v_{th}$ with the same fractional bits (i.e., 4 bits) still maintains the accuracy compared to the baseline, as shown by label-③ and label-④ for the PTQ and the ITQ, respectively. The reason is that the values for updating the $v$ and $v_{th}$ can be represented using fewer fractional bits than the ones for updating the weights $w$. These also indicate that the weights are more significant than the neuron parameters, as their small update can change the accuracy significantly. Hence, quantizing all parameters of the U-SNN also leads to a notable accuracy degradation when the fractional bitwidth is reduced to 4 bits (or fewer), as pointed by label-⑤ and label-⑥ for the PTQ and the ITQ, respectively

**Accuracy of the Supervised SNN:** In the S-SNN case, we quantize the weights ($w$), and the neuron parameters ($\alpha$, $\beta$, $\gamma$, $P$, $Q$, $R$, and $v$), and the experimental results are presented in Figure 4.28. Here, N($Qi.f$) represents the precision of variables $P$, $Q$, $R$, and $v$. Notable accuracy degradation from the baseline accuracy is observed when reducing the fractional bits of the weights and/or the neuron parameters to 10 bits (and fewer), indicating that weights and neuron parameters have comparable significance to the accuracy. These also indicate that the S-SNN requires considerable bitwidth to maintain the high accuracy for the DVS-Gesture dataset. The reason is that, the DVS-Gesture is a relatively complex dataset because, besides considering the stream of events in each

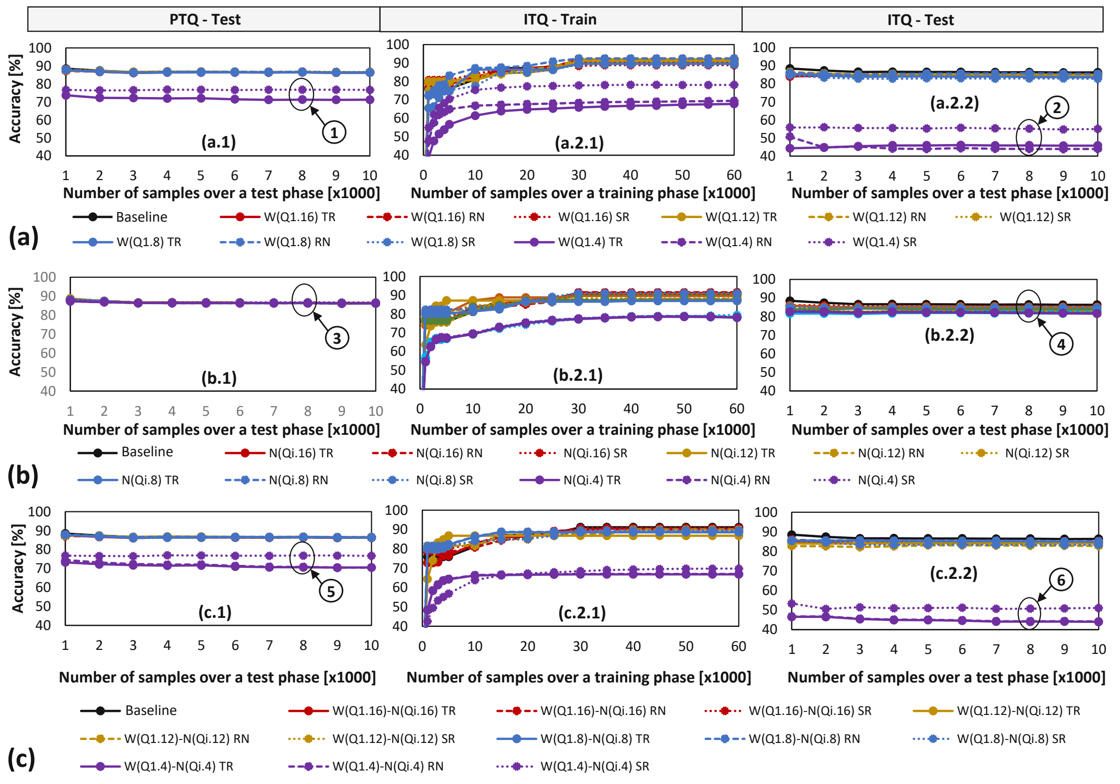


Figure 4.27: Results for the U-SNN with the MNIST dataset, when varying (a) the precision of weights and the rounding schemes, (b) the precision of neuron parameters and the rounding schemes, (c) the precision of weights and neuron parameters, as well as the rounding schemes. Labels (a.1/b.1/c.1) indicate the test accuracy in the PTQ, (a.2.1/b.2.1/c.2.1) indicate the estimated accuracy during the training in the ITQ, and (a.2.2/b.2.2/c.2.2) indicate the test accuracy in the ITQ.

frame, the network has to draw a correct conclusion of a gesture for the complete stream of events from multiple frames. Hence, it requires considerable bitwidth to distinguish a gesture from other gestures in each frame and in a complete stream of events.

**Additional Discussion:** We also make the following observations across different network types (i.e., U-SNN and S-SNN) and different quantization approaches.

- The SR scheme generally achieves slightly better accuracy than other rounding schemes, because this scheme is not biased towards a specific rounding direction, thereby having a higher probability of values that lead to higher accuracy. However, it consumes the highest hardware resource as it needs a random number generator.
- Different combinations of the quantization and rounding schemes achieve various accuracy, but their values are not significantly different. Users can decide the quantization and rounding schemes, as well as the parameters to be quantized, that are suitable for the target applications, considering the accuracy and memory constraints, and the exploration cost. Therefore, the overhead depends on the selected scheme.

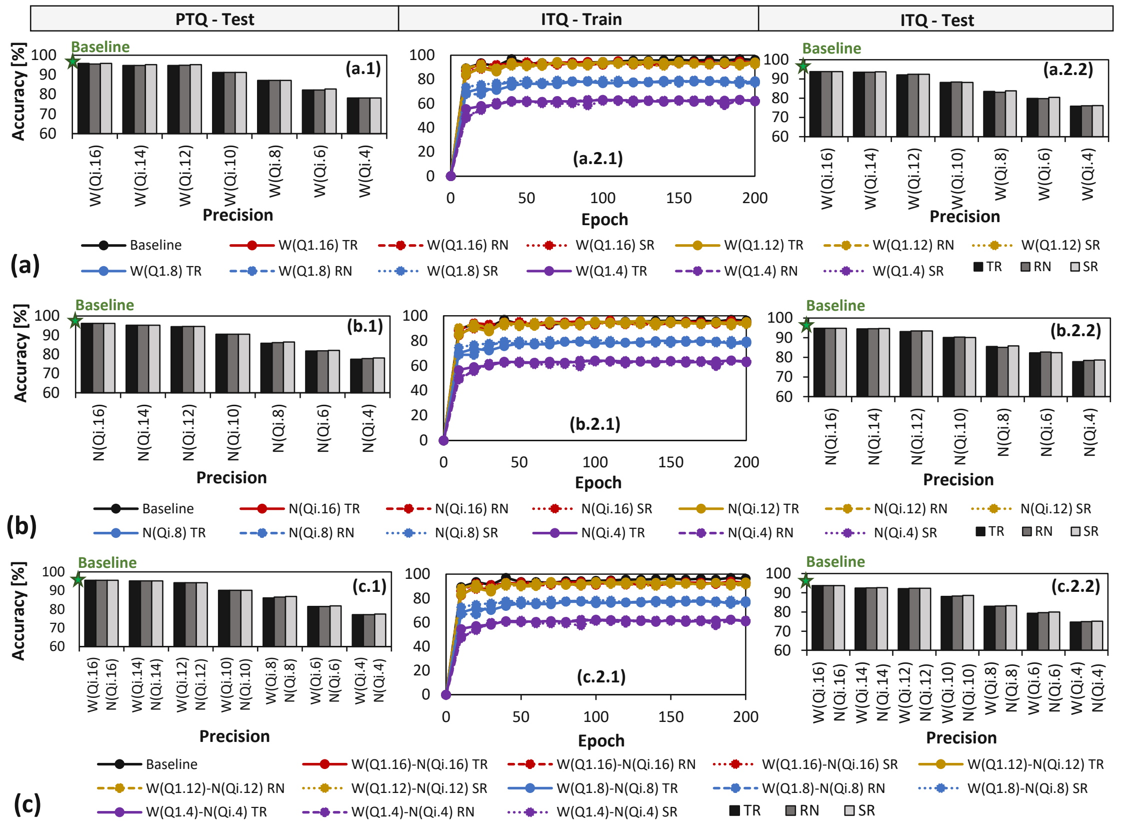


Figure 4.28: Results for the S-SNN with the DVS-Gesture dataset, when varying (a) the precision of weights and the rounding schemes, (b) the precision of neuron parameters and the rounding schemes, (c) the precision of weights and neuron parameters, as well as the rounding schemes. Labels (a.1/b.1/c.1) indicate the test accuracy in the PTQ, (a.2.1/b.2.1/c.2.1) indicate the estimated accuracy during the training in the ITQ, and (a.2.2/b.2.2/c.2.2) indicate the test accuracy in the ITQ.

### 4.3.6.2 SNN Model Selection with Memory-Accuracy Trade-Offs

To find the SNN model that offers a good memory-accuracy trade-off, we employ the proposed reward function in Equation 4.4 that quantifies the trade-off benefit of the given model. To do this, we need to define the coefficient $\mu$ in the reward function. Small $\mu$ means that the function prioritizes the weight of accuracy more than the memory. On the other hand, large $\mu$ means that the function prioritizes the weight of memory more than the accuracy. The users can define the value of $\mu$ based on their preferences to meet the design specifications. In this work, for the exploration purpose, we define the value of coefficient $\mu \in \{0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1\}$.

**Model Selection for the Unsupervised SNN:** We apply the proposed reward function to the U-SNN model candidates and the results are provided in Figure 4.29(a), from which we make the following observations.

- Label-❶: The model that has the highest reward for $\mu = 0.01$ is the one that employs W($Q1.8$)-N($Qi.8$) precision, and achieves 86.6% accuracy and 3.2x memory saving.
- Label-❷: The model that has the highest reward for $\mu = 0.3$ is the one that employs W($Q1.6$)-N(FP32) precision, and achieves 86.3% accuracy and 3.9x memory saving.
- Label-❸: The model that has the highest reward for $\mu \in \{0.1, 0.2, 0.4, 0.5, 1\}$ is the one that employs W($Q1.6$)-N($Qi.6$) precision, and achieves 86.2% accuracy and about 4x memory saving.

These results show that larger $\mu$ shifts the preferred model towards the one with smaller memory, which typically has lower accuracy. Meanwhile, smaller $\mu$ shifts the preferred model towards the one with higher accuracy, which typically has a larger memory footprint. If the maximum tolerance of accuracy degradation is 1% from the baseline, then the model with W($Q1.6$)-N($Qi.6$) precision is the Pareto-optimal one, with 86.2% accuracy and about 4x memory saving.

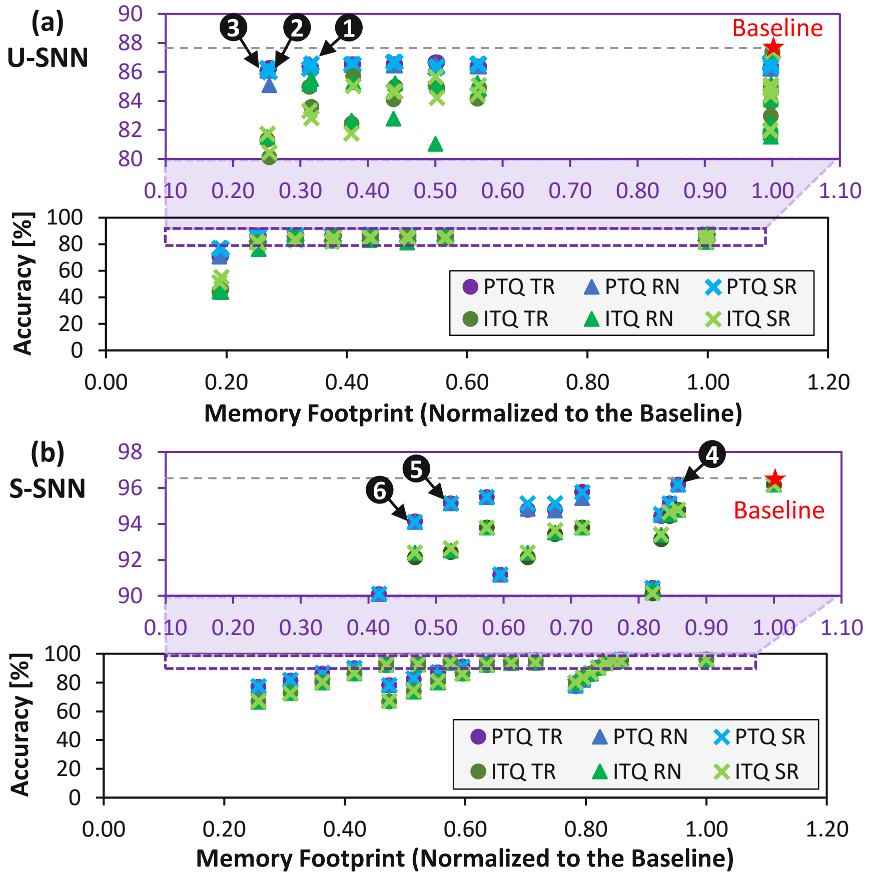


Figure 4.29: Accuracy vs. normalized memory footprint for (a) U-SNN and (b) S-SNN.

**Model Selection for the Supervised SNN:** We also apply the proposed reward function to the S-SNN model candidates and the results are provided in Figure 4.29(b), from which we make the following observations.

- Label-❹: The model that has the highest reward for $\mu = 0.01$ is the one that employs W(FP32)-N($Qi$.16) precision, and achieves 96.1% accuracy and 1.2x memory saving.
- Label-❺: The model that has the highest reward for $\mu = 0.1$ is the one that employs W($Q$1.14)-N($Qi$.14) precision, and achieves 95.1% accuracy and 1.9x memory saving.
- Label-❻: The model that has the highest reward for $\mu \in \{0.2, 0.3, 0.4, 0.5, 1\}$ is the one that employs W($Q$1.12)- N($Qi$.12) precision, and achieves 94.1% accuracy and 2.1x memory saving.

Here, a similar trend regarding the impact of $\mu$ is also observed, e.g., larger $\mu$ shifts the preferred model towards the one with smaller memory and lower accuracy. If the maximum tolerance of accuracy degradation is only 1% from the baseline, then the model with W(FP32)-N($Qi$.16) precision level is selected. If we relax the tolerance to 2%, it suggests a different Pareto-optimal SNN model, i.e., the model with W($Q$1.14)-N($Qi$.14) precision, 95.1% accuracy, and 1.9x memory saving. If we relax the tolerance even more, e.g., allowing 3% accuracy degradation, then the one with W($Q$1.12)-N($Qi$.12) precision (with 94.1% accuracy and 2.1x memory saving) is selected.

The above results and discussion show that our Q-SpiNN framework provides (1) comprehensive information about the accuracy and the memory of the given SNN models under different quantization approaches, and (2) an effective model selection to find the efficient SNN model. Moreover, the users can set $\mu$ with their preferred value in the reward function to select an SNN model that meets the design requirements.

### 4.3.7 Summary of Q-SpiNN Framework

We propose the Q-SpiNN framework for quantizing SNNs through (1) quantization of different parameters, (2) exploration that considers different quantization schemes, precision levels, and rounding schemes, and (3) employment of a reward function for model selection. For the unsupervised SNN, the Q-SpiNN obtains about 4x memory saving and keeps the accuracy within 1% from the baseline on the MNIST dataset. For the supervised one, it obtains about 2x memory saving and keeps the accuracy within 2% from the baseline on the DVS-Gesture dataset. Therefore, our framework enables SNNs to be deployable on many AI applications under tight memory budgets.

## 4.4 EnforceSNN: Energy-Efficient SNN Inference with Approximate DRAMs

This section aims at addressing **Problem-3** with the solution for substantially decreasing the DRAM access energy of SNN hardware platforms for the inference phase through HW-level approximation in DRAM, while maintaining accuracy.

### 4.4.1 Motivational Study

To enable the full potential of DRAM energy savings, the effective optimization should jointly minimize the DRAM energy-per-access and the number of DRAM accesses, by leveraging the approximation in DRAM to expose the full energy-saving potential. To highlight the potential of reduced-voltage-based approximate DRAM, we perform an experimental case study. Here, we aim at observing (1) the dynamics of DRAM bitline voltage ($V_{bitline}$) for both the accurate and approximate DRAM settings, and (2) the DRAM access energy for different access conditions (including a row buffer hit, miss, or conflict). Note, $V_{bitline}$ is defined as the voltage measured in each DRAM bitline when a DRAM supply voltage ($V_{supply}$) is applied.

In the experiments, we employ the DRAM circuit model from previous work [CYG+17a] and the SPICE simulator to study the dynamics of $V_{bitline}$. The accurate DRAM operates at 1.35 V of the supply voltage ($V_{supply}$), while the approximate one operates at 1.025 V. Further details on the experimental setup are discussed in Section 4.4.6. We consider the LPDDR3-1600 4Gb DRAM configuration as it is representative of the low-power DRAM types for embedded systems. We also employ the DRAMPower simulator to estimate the DRAM access energy because it has been validated against real measurements [Cha14] and has been widely used in the computer architecture communities. For the network, we consider the FC-based SNN architecture in Figure 4.8(a). The experimental results are presented in Figure 4.30, from which we make the following key observations.

- The reduced-voltage DRAM decreases the DRAM energy-per-access across different access conditions, i.e., by up to 42% of energy reduction for each access; see Figure 4.30(a)-(b).
- The row buffer hit has lower energy consumption than the row buffer miss or conflict; see Figure 4.30(b). Moreover, the row buffer hit also incurs less latency than the row buffer miss or conflict [PHS20, PHS21b]. Therefore, the row buffer hit should be exploited to optimize the DRAM latency and energy.
- The $V_{bitline}$ decreases as the $V_{supply}$ decreases, hence forcing the DRAM cells to operate under lower reliability as the *weak cells* may fail to hold the correct bits. *Weak cells* are DRAM cells that fail when the DRAM parameters (e.g., voltage) are reduced [CYG+17a, KPHM18]. This condition is indicated by the increase of bit error rate (BER) on DRAM as the $V_{supply}$ decreases; see Figure 4.30(c)

Moreover, the reduced-voltage approximate DRAM technique can also be combined with state-of-the-art techniques to further improve the energy efficiency of SNN inference. For example, Figure 4.30(d) shows the estimated DRAM energy savings achieved by our technique when combined with the weight pruning.

## 4.4.2 Scientific Research Challenges

Although employing the approximate DRAM can substantially decrease the DRAM energy-per-access, it also decreases the DRAM reliability since the bit errors increase when the $V_{supply}$ decreases, as shown in Figure 4.30(c). These errors may degrade the accuracy of SNN inference since they can change the weight values in DRAM, which then deteriorates the neuron behavior. Therefore, the key challenge is *how to achieve low DRAM access energy for SNN inference using approximate DRAM, while minimizing the negative impact of DRAM errors on the accuracy.*

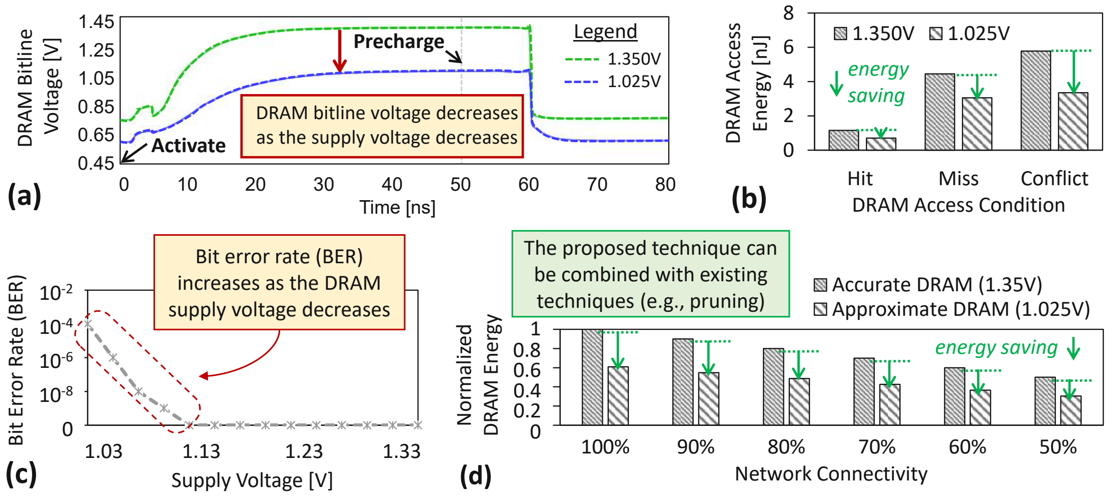


Figure 4.30: (a) The dynamics of $V_{bitline}$ under different $V_{supply}$ values. (b) DRAM access energy for a row buffer hit, a row buffer miss, and a row buffer conflict, under different $V_{supply}$ values. (c) BER of approximate DRAM and its respective supply voltage $V_{supply}$; adapted from studies in [CYG+17a]. (d) The estimated potential of DRAM energy savings achieved by our technique when combined with the weight pruning across different rates of network connectivity (i.e., synaptic connections) for a network with 4900 excitatory neurons. The results are obtained from experiments using the LPDDR3-1600 4Gb DRAM configuration and the DRAMPower simulator [Cha14].

***Required:*** *A systematic approach for reducing the DRAM supply voltage of the SNN hardware platforms, thereby substantially decreasing the system energy during inference while mitigating the negative impact of approximation errors.*

### 4.4.3 Novel Contributions

To overcome the above research challenges, we propose *EnforceSNN, a novel framework that enables resilient and energy-efficient SNNs considering approximate DRAMs (i.e., reduced-voltage DRAMs) for embedded systems* [PHS22a]. Based on the best of our knowledge, this work is the first effort that employs approximate DRAM for improving the energy efficiency of SNN inference, while enhancing their error tolerance against bit errors in DRAM. Our EnforceSNN framework employs the following key steps.

1. **Employing weight quantization** to reduce the memory footprint for SNN weights and the number of DRAM accesses for SNN inference, thereby optimizing the DRAM access energy.
2. **Devising an efficient DRAM data mapping** to maximize row buffer hits for optimizing the DRAM energy-per-access while considering BER in DRAM.
3. **Analyzing the SNN error tolerance** to understand the SNN accuracy profile under different DRAM supply voltage values and different BER values.

4. **Improving the SNN error tolerance** by developing and employing efficient fault-aware training (FAT) that considers SNN accuracy profile and bit error locations in DRAM.

5. **Devising an algorithm to select the SNN model** that offers good trade-offs among accuracy, memory, and energy consumption from the given model candidates using the proposed reward function.

### 4.4.4 Approximate DRAM

#### Dynamics of the Reduced-Voltage DRAM

To understand the dynamics of the reduced-voltage DRAM, we perform extensive experiments using the SPICE simulator and the DRAM circuit model from [CYG+17a] while considering different supply voltage ($V_{supply}$) values, to characterize the parameters of reduced-voltage DRAM, i.e, including the bitline voltage ($V_{bitline}$) and the respective timing parameters (i.e., $t_{RP}$, $t_{RCD}$, and $t_{RAS}$). The experimental results are presented in Figure 4.31, and the obtained parameters are used for further DRAM energy estimation. The ready-to-access voltage is defined as the condition when $V_{bitline}$ reaches 75% of $V_{supply}$, which represents the minimum $t_{RCD}$ for reliable DRAM operations, as shown by Ⓐ. The ready-to-precharge voltage is defined as the condition when $V_{bitline}$ reaches 98% of $V_{supply}$, which represents the minimum $t_{RAS}$ for reliable DRAM operations, as shown by Ⓑ. Meanwhile, the ready-to-activate voltage is defined as the condition when the $V_{bitline}$ is within 2% of $V_{supply}/2$, which represents the minimum $t_{RP}$ for reliable DRAM operations, as shown by Ⓒ.

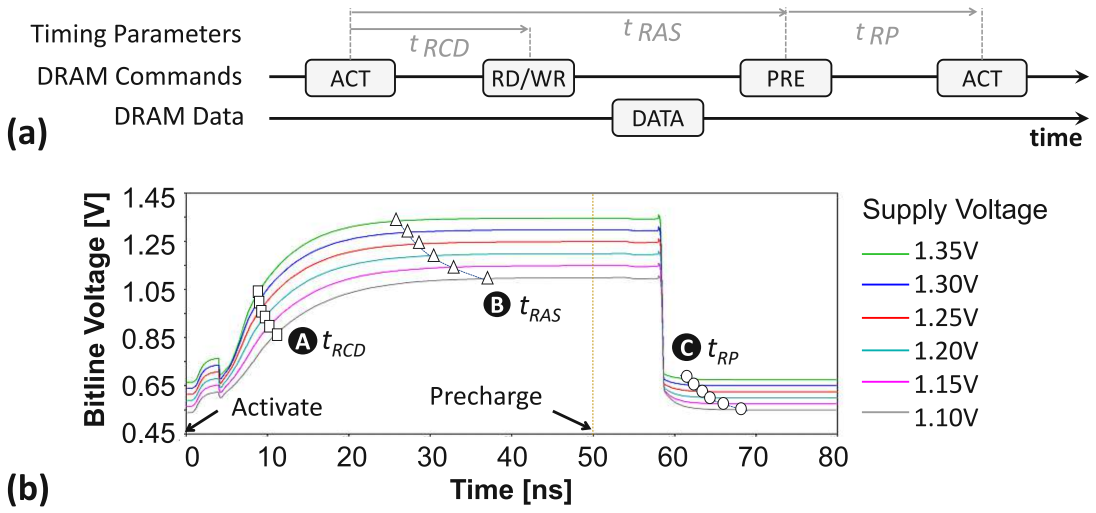


Figure 4.31: (a) Diagram of the DRAM commands (i.e., ACT, RW or WR, and PRE) and the DRAM timing parameters (i.e., $t_{RCD}$, $t_{RAS}$, and $t_{RP}$). (b) The dynamics of the DRAM bitline voltage $V_{bitline}$ and the respective timing parameters.

### Error Modeling for Approximate DRAM

To obtain accurate error profiles of commercial DRAMs under reduced-voltage settings, a complex hardware experimental platform that can apply different voltage settings and capture the map of approximation-induced errors, is required. However, this approach requires costly resources as well as long design and investigation time. To address these limitations, previous work [KOY+19] has proposed four error models that closely fit the real reduced-voltage-based approximate DRAMs as the following.

- **Error Model-0:** The bit errors have a *uniform random distribution* across a DRAM bank. The errors are modeled by considering (1) the *weak cells*, i.e., cells that fail when the DRAM parameters are reduced, and (2) the probability of an error in any weak cell.
- **Error Model-1:** The bit errors have a *vertical distribution* across the bitlines of a DRAM bank. The errors are modeled by considering (1) the weak cells in bitline $B$, and (2) the probability of an error in the weak cells of bitline $B$.
- **Error Model-2:** The bit errors have a *horizontal distribution* across the wordlines of a DRAM bank. The errors are modeled by considering (1) the weak cells in wordline $W$, and (2) the probability of an error in the weak cells of wordline $W$.
- **Error Model-3:** It is a data-dependent error model, i.e., the profile of the bit errors follows a *uniform random distribution that depends on the content of the DRAM cells*. The errors are modeled by considering (1) the weak cells, (2) the probability of an error in the weak cells that contain a 1 value, and (3) the probability of an error in the weak cells that contain a 0 value.

In this work, we employ the **Error Model-0**, due to the following reasons: (1) it produces errors with high similarity to the real reduced-voltage-based approximate DRAM by using the percentage of weak cells and the error probability in any weak cell; (2) it offers a reasonable approximation of other error models, including the approximation of (a) errors across bitlines similar to Error Model-1, (b) errors across wordlines similar to Error Model-2, and (c) uniform random distribution similar to Error Model-3; and (3) it provides fast error injection through software. Previous work [KOY+19] also employed the DRAM Error Model-0 majorly due to similar reasons.

### 4.4.5 EnforceSNN Framework

The EnforceSNN framework employs the following key mechanisms for enabling a resilient and energy-efficient SNN inference under the presence of voltage-induced DRAM errors (the overview is shown in Figure 4.32).

1. **Quantizing of the SNN weights (Section 4.4.5.1).** It aims at reducing the memory footprint of SNN weights which leads to the reduction of DRAM accesses, and hence DRAM access energy for SNN inference. The quantization is performed using truncation.

2. **Devising an error-aware DRAM data mapping policy (Section 4.4.5.2)** to optimize the DRAM energy-per-access. The proposed policy considers (1) mapping the data in the appropriate DRAM part (e.g., subarray) whose error rate meets the BER requirement, and (2) maximizing the row buffer hits and exploiting the multi-bank burst feature.
3. **Analyzing the SNN error tolerance (Section 4.4.5.3)**. It aims at understanding the accuracy profile of SNN inference considering different BER values. The idea is to classify the range of BER values based on their impact on accuracy. This information is then leveraged for developing an efficient fault-aware training (FAT) technique.
4. **Improving the SNN error tolerance (Section 4.4.5.4)** through an efficient FAT technique that considers the selected BER values from the SNN error-tolerance analysis. Hence, SNN error tolerance is improved under reduced training time and energy consumption.
5. **Developing an SNN model selection algorithm (Section 4.4.5.5).** Its target is to find an appropriate model from the given candidates considering their accuracy, memory, and energy consumption. Our algorithm quantifies the trade-off benefits of the candidates using our multi-objective reward function, then selects the model with the highest benefit.

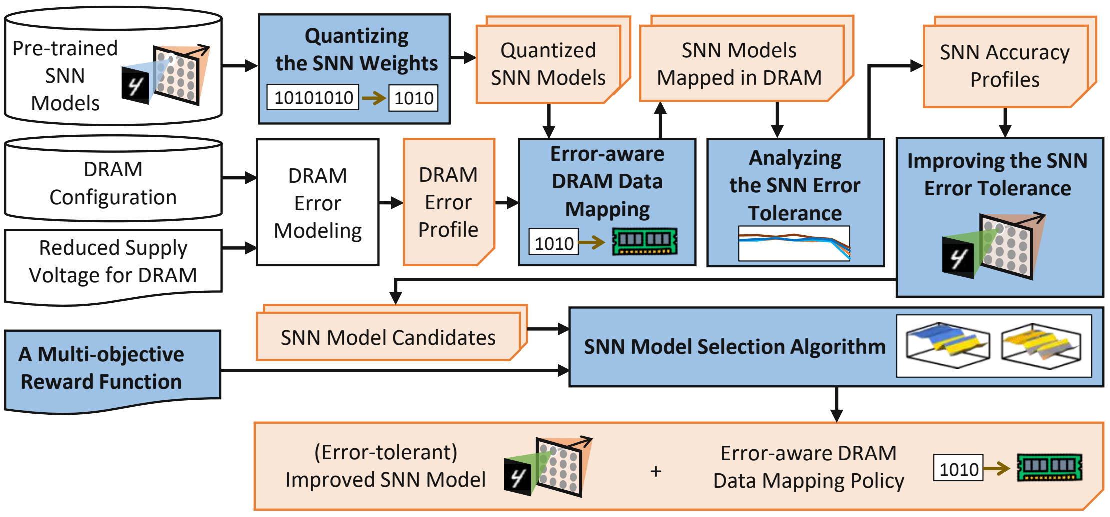

Pre-trained SNN Models
Quantizing the SNN Weights
10101010
1010
Quantized SNN Models
SNN Models Mapped in DRAM
SNN Accuracy Profiles
DRAM Configuration
DRAM Error Modeling
DRAM Error Profile
Error-aware DRAM Data Mapping
1010
Analyzing the SNN Error Tolerance
Improving the SNN Error Tolerance
Reduced Supply Voltage for DRAM
SNN Model Candidates
SNN Model Selection Algorithm
A Multi-objective Reward Function
(Error-tolerant) Improved SNN Model
+
Error-aware DRAM Data Mapping Policy
1010

Figure 4.32: An overview of the EnforceSNN framework. The novel contributions are highlighted in blue.

### 4.4.5.1 Quantizing the SNN Weights

*We employ weight quantization to substantially reduce the memory footprint of the SNN model and the number of DRAM accesses, which lead to DRAM energy saving.* The reason is that, quantization is a prominent technique for reducing the memory footprint of NNs [GAGN15, MNA$^+$18]. Moreover, *this work is the first effort to study and exploit*

*SNN weight quantization considering approximation errors in DRAM, thereby providing new insights as compared to previous studies on SNN weight quantization.* Our weight quantization considers the fixed-point format which can be represented as $Qi.f$ (see the quantization concept in Section 2.5). Note, the fixed-point format $Qi.f$ can also be represented as FxP(1+$i$+$f$). We select the "signed $Qi.f$" format to show that our EnforceSNN framework provides a generic solution with high applicability for different variants of bio-plausible learning rules (e.g., STDP variants) which may lead to positive or negative weight values [RAIAS+14, DC15]. To do this, we perform a fixed-point quantization to the trained SNN weights using a specific rounding scheme, such as truncation (TR), round to the nearest (RN), or stochastic rounding (SR). For a case study, we select the truncation as it provides competitive accuracy with low computational complexity [PS21a, PS22a, PS22b]. To illustrate this, we evaluate the impact of different rounding schemes on the accuracy through an experimental case study, and the results are shown in Figure 4.33. Since TR keeps the $f$ bits and removes the other fractional bits, the output fixed-point for the given real number $x$ with $Qi.f$ format is defined as $TR(x, Qi.f) = \lfloor x \rfloor$. In our SNN model, we employ the pair-based weight-dependent STDP learning rule (from Section 2.3.3) that bounds each weight value ($w$) within the defined range, i.e., $w = [0, 1]$. Consequently, applying the truncation to the weights will round the value down. In this work, we consider an 8-bit fixed-point with "signed $Q1.6$" and 2's complement format (i.e., 1 sign bit, 1 integer bit and 6 fractional bits), since it provides high accuracy for SNNs under unsupervised learning scenarios [PS21a]. Note, we can also employ the "unsigned $Qi.f$" format without sign bit to represent the 8-bit non-negative weights (i.e., 1 integer bit and 7 fractional bits) in the EnforceSNN if desired. For both "signed $Qi.f$" and "unsigned $Qi.f$" formats, 1 bit for the integer part is required for representing the maximum possible weight value (i.e., $w = 1$).

**Quantization Steps:** We quantize only the weights through the simulated quantization approach, which represents the weight values under fixed-point format, and perform computations under floating-point format [Kri18, JKC+18b, GKD+, vBKM+22]. To perform quantization, we convert the weight values from 32-bit floating-point format (FP32) to 8-bit fixed-point format (signed $Q1.6$) by constructing their 8-bit binary representations under 32-bit integer format (INT32), thereby conveniently performing bit-wise modification and rounding operation while considering the sign and the rounding scheme (i.e., truncation). Afterward, we convert the quantized weight values (INT32) to FP32 format through casting and then normalizing the values by $2^f$. Hence, the 8-bit binary representations of quantized weight values are saved in FP32 and can be used in the FP32-based arithmetic computations.

### 4.4.5.2 Error-aware DRAM Data Mapping Policy

**DRAM Error Injection:** *If there is no DRAM error*, the quantization steps are performed, and the quantized weight values (in FP32) can be used for computations in SNN processing. *If DRAM errors exist*, the quantization steps are performed while considering the DRAM error injection. These errors are injected to the 8-bit binary

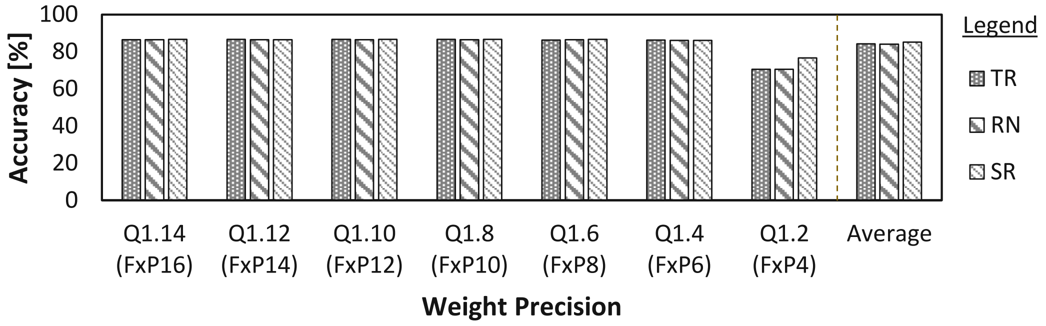


Figure 4.33: The accuracy of a 900-neuron network on the MNIST across different precision levels and rounding schemes, i.e., truncation (TR), round to the nearest (RN), and stochastic rounding (SR). These results show that employing TR on top of the FxP8 precision leads to competitive accuracy than other rounding schemes (i.e., RN and SR).

representations of quantized weights (in INT32) under a specific DRAM data mapping policy. Afterward, we convert the binary representations of quantized weights (in INT32) to FP32 format, so that the quantized weight values can be used for computations in SNN processing.

**Proposed Data Mapping Policy:** It is important to map the SNN model properly in DRAM to ensure that (1) the weights are minimally affected by errors in DRAM so that the accuracy is maintained, and (2) the DRAM energy-per-access is optimized. Towards this, *we devise and employ an error-aware DRAM mapping policy to place the SNN weights in DRAM, while optimizing the DRAM energy-per-access.* The proposed DRAM mapping policy is illustrated in Figure 4.34(a), and its key ideas are explained in the following.

1. The weights are mapped in the appropriate DRAM part (e.g., chip, bank, or subarray), whose error rate meets the BER requirement, i.e., $\leq$ the maximum tolerable BER ($BER_{th}$). Here, we consider the subarray-level granularity for data mapping, since it allows us to exploit the following features.

   - *The multi-bank burst feature*, which is available in commodity DRAMs, can be employed to increase the throughput. Its timing diagram is shown in Figure 4.34(b).
   - *The subarray-level parallelism*, which is available in novel DRAM architectures (e.g., SALP [KSL$^+$12]), can also be employed to increase the throughput.

   We determine $BER_{th}$ through experiments that investigate the accuracy profile of a network across different BER values. Figure 4.35 shows the experimental results for a 900-neuron network. If the accuracy scores are significantly lower than the baseline accuracy without bit errors (i.e., >3% accuracy degradation), we refer the respective BER values as *intolerable BER*, as shown by ① and ② in Figure 4.35. Otherwise, we define them as *tolerable BER*. For instance, ③ in Figure 4.35 shows an accuracy with a tolerable BER. Based on this discussion, we define $BER_{th} = 10^{-2}$.

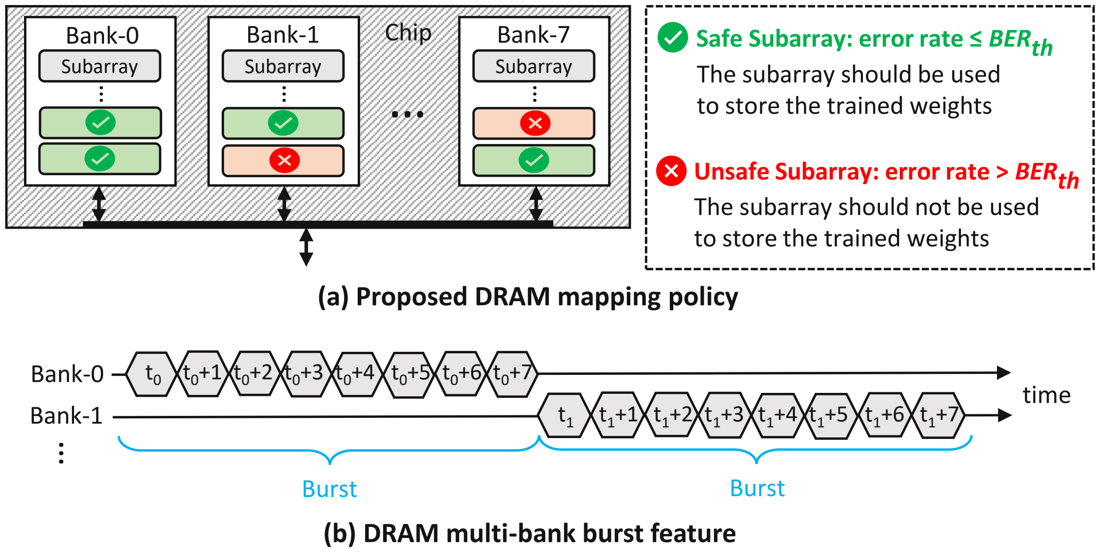

Bank-0
Subarray
Bank-1
Subarray
Chip
Bank-7
Subarray
Safe Subarray: error rate ≤ $BER_{th}$
The subarray should be used to store the trained weights
Unsafe Subarray: error rate > $BER_{th}$
The subarray should not be used to store the trained weights
(a) Proposed DRAM mapping policy
Bank-0
Bank-1
$t_0$ $t_0$+1 $t_0$+2 $t_0$+3 $t_0$+4 $t_0$+5 $t_0$+6 $t_0$+7
$t_1$ $t_1$+1 $t_1$+2 $t_1$+3 $t_1$+4 $t_1$+5 $t_1$+6 $t_1$+7
time
Burst
Burst
(b) DRAM multi-bank burst feature

Figure 4.34: (a) Our proposed DRAM mapping policy, leveraging subarray-level granularity. (b) The timing diagram of the DRAM multi-bank burst feature.

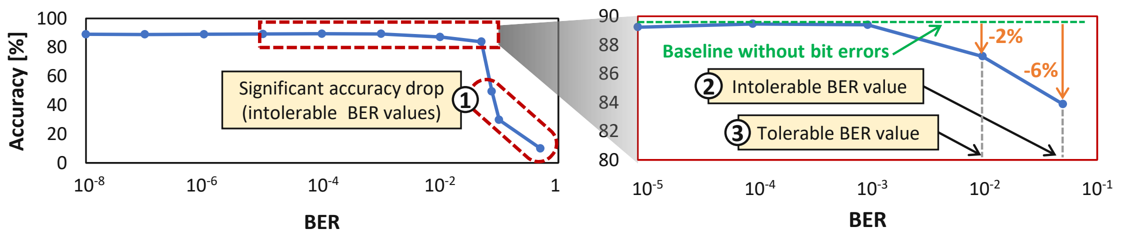


Figure 4.35: The test accuracy profile of a 900-neuron network across different BER values, showing the tolerable and intolerable BER. This network has a fully-connected architecture like in Figure 2.12(a) where each input pixel is connected to all neurons, and the output of each neuron is connected to other neurons for performing inhibition, thereby providing competition among neurons to correctly recognize the input class.

2. The weights are mapped in a way to maximize the row buffer hits for optimizing the DRAM energy-per-access while exploiting the multi-bank burst feature for maximizing the data throughput. The reason is that a row buffer hit incurs the lowest DRAM access energy than other access conditions (i.e., a row buffer miss or conflict), as suggested by our experimental results in Figure 4.30(b).

To efficiently implement the above ideas, we devise Algorithm 7 with the following steps. First, we identify the subarrays whose error rate $\leq BER_{th}$ and refer them as *the safe subarrays*, which should be used for storing the weights. Otherwise, we refer the subarrays whose error rate $> BER_{th}$ as *the unsafe subarrays*, which should not be used for storing the weights. This step is represented in line 7 of Algorithm 7. Second, to maximize

**Algorithm 7** The proposed DRAM mapping policy

```
INPUT: (1) DRAM ($DRAM$): number of channel ($n_{ch}$), number of rank-per-channel ($n_{ra}$),
    number of chip-per-rank ($n_{cp}$), number of bank-per-chip ($n_{ba}$), number of subarray-per-bank
    ($n_{su}$), number of row-per-subarray ($n_{ro}$), number of column-per-row ($n_{co}$);
    (2) Bit error rate (BER): BER of a subarray ($BER\_subarray$), maximum tolerable BER
    ($BER_{th}$);
    (3) Data ($data$);
OUTPUT: DRAM ($DRAM$);
BEGIN
    Process:
1:  for $ch = 0$ to $(n_{ch} - 1)$ do
2:    for $ra = 0$ to $(n_{ra} - 1)$ do
3:      for $cp = 0$ to $(n_{cp} - 1)$ do
4:        for $ro = 0$ to $(n_{ro} - 1)$ do
5:          for $su = 0$ to $(n_{su} - 1)$ do
6:            for $ba = 0$ to $(n_{ba} - 1)$ do
7:              if $BER\_subarray[ch, ra, cp, ba, su] \leq BER_{th}$ then
8:                for $co = 0$ to $(n_{co} - 1)$ do
9:                  $DRAM[ch, ra, cp, ba, su, ro, co] \leftarrow data$;
10: return $DRAM$;
END
```

the row buffer hits and exploit the multi-bank burst feature, the data mapping in each DRAM chip should follow the following policy (represented in lines 3-8 of Algorithm 7).

- **Step-1:** Assume that we consider mapping data to a target subarray of the target bank with the following initial indices, i.e., *subarray_index = i, bank_index = j.*
- **Step-2:** If the target subarray is a safe subarray, then we prioritize mapping the data to different columns in the same row for maximizing row buffer hits. Otherwise, this subarray is not utilized and we move to another target subarray in a different bank (*subarray_index = i, bank_index += 1*) to exploit bank-level parallelism. Then, we perform **Step-2** again. If all columns in the same row across all banks are fully filled or unavailable, then we move to another subarray in the initial bank (*subarray_index += 1, bank_index = j*) to exploit subarray-level parallelism, if applicable.
- **Step-3:** In the target subarray, the remaining data are mapped in the same fashion as **Step-2**. When all columns in the same row in all safe subarrays across all banks are fully filled, then the remaining data are placed in a different row of the initial target subarray and bank (*subarray_index = i, bank_index = j*). Afterward, we perform **Step-2** to **Step-3** again until all data are mapped to a DRAM chip. If some data still remain but there are no available spaces in a DRAM chip, then we perform **Step-4**.
- **Step-4:** The remaining data are mapped using **Step-1** to **Step-3** to different DRAM chips, ranks, and channels respectively, if applicable.

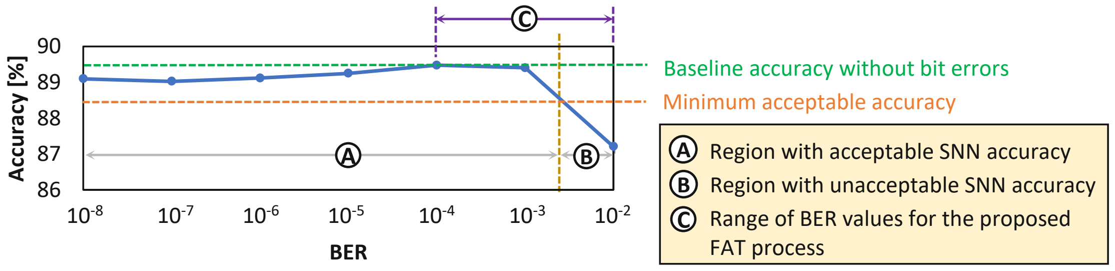


Figure 4.36: The test accuracy profile of a 900-neuron network shows the region with acceptable accuracy, the region with unacceptable accuracy, and the range of BER values for the proposed fault-aware training. Note, this network is the same as the one in Figure 4.35. Here, the observation focuses on the BER values that can be considered in the retraining process, thereby having a smaller range than the one in Figure 4.35.

#### 4.4.5.3 Analyzing the SNN Error Tolerance

Previous discussion highlights that bit errors in the SNN weights can degrade the accuracy, as they change the weight values and deviate the neuron behavior from the correct classification. Therefore, SNN error tolerance should be improved, so that the SNN model can achieve high accuracy even in the presence of high error rate. To effectively enhance the SNN error tolerance, it is important to understand the SNN accuracy profile under DRAM errors. Towards this, *our EnforceSNN analyzes the accuracy profile of the SNN model considering the data mapping pattern in DRAM and different BER values.* We observe that the accuracy profile typically has acceptable accuracy (i.e., within 1% accuracy degradation from the baseline without errors) when BER is low, and has notable accuracy degradation when BER is high. Therefore, we classify the accuracy profile into two regions: Ⓐ a region with acceptable accuracy, and Ⓑ a region with unacceptable accuracy, as shown in Figure 4.36. These insights will be leveraged for developing an efficient enhancement technique for improving the SNN error tolerance in Section 4.4.5.4.

#### 4.4.5.4 Improving the SNN Error Tolerance

*Our EnforceSNN enhances the SNN error tolerance through a fault-aware training (FAT) technique that incorporates the error profile of the approximate DRAM.* We consider to efficiently perform FAT for minimizing training time, energy consumption, and carbon emission [SGM19, SGM20], by conducting a small yet effective number of iterations for the retraining process, while avoiding *accuracy collapse*. Accuracy collapse is a significant accuracy degradation due to training divergence that is caused by introducing high BER immediately at the beginning of the retraining process [KOY+19]. The proposed FAT technique has the following key steps, which are also presented in Algorithm 8.

- **Step-1:** We define the range of BER values that will be incorporated in the training process to make the SNN model adaptable to DRAM errors, as shown by region-Ⓒ in Figure 4.36. We incorporate (1) BER values in region-Ⓐ that are close to region-Ⓑ, and (2) all BER values in region-Ⓑ, in the training process. Here, we consider the

**Algorithm 8** The proposed FAT technique

```
INPUT: (1) Baseline pre-trained SNN: model ($model_0$), accuracy ($model_0.acc$);
    (2) DRAM error model ($DRAMerr$);
    (3) BER for retraining: error rates ($BER$), number of error rates ($N_{BER}$);
    (4) Training dataset: samples ($S_{train}$), number of samples ($N_{train}$);
    (5) Test dataset: samples ($S_{test}$), number of samples ($N_{test}$);
OUTPUT: (1) Improved SNN: model ($model_1$), accuracy ($model_1.acc$);
BEGIN
    Initialization:
 1: $model_{temp} = model_0$;
 2: $model_1 = model_0$;
 3: $model_1.acc = 0$;
    Process:
 4: for $i = 0$ to $(N_{BER} - 1)$ do
 5:     $error\_map = DRAMerr(BER[i])$; // error generation
 6:     inject $error\_map$ into $model_{temp}$; // error injection
 7:     for $r = 0$ to $(N_{train} - 1)$ do
 8:         train $model_{temp}$ with $S_{train}[r]$; // train
 9:     for $s = 0$ to $(N_{test} - 1)$ do
10:         test $model_{temp}$ with $S_{test}[s]$; // test
11:     if $model_{temp}.acc > model_1.acc$ then
12:         $model_1 = model_{temp}$;
13:         $model_1.acc = model_{temp}.acc$;
14: return $model_1$;
END
```

two highest BER values in region-Ⓐ in the training process to make the model adapt to high fault rates safely with less training time, without facing accuracy collapse.

- **Step-2:** The bit errors in DRAM are generated for different BER values (which correspond to different $V_{supply}$ values), based on the DRAM error model-0 that follows a uniform random distribution across a DRAM bank.

- **Step-3:** The generated bit errors are then injected into the DRAM cell locations, and the weight bits in these locations are flipped. In this step, we consider the proposed DRAM data mapping discussed in Section 4.4.5.2 for maximizing the row buffer hits and exploiting the multi-bank burst feature.

- **Step-4:** We include the generated bit errors in the retraining process by incrementally increasing the BER from the minimum rate to a maximum one following the defined range of BER values from **Step-1**. We increase the BER value after each epoch of retraining by a defined ratio (e.g., 10x from the previous error rate). In this manner, the SNN model is gradually trained to tolerate DRAM errors from the defined lowest rate to the maximum one, thereby carefully improving the SNN error tolerance.

#### 4.4.5.5 Algorithm for SNN Model Selection

From the previous steps, we may get different sizes of error-tolerant SNN models as potential solutions for the given embedded applications. Therefore, we need to consider design trade-offs to select the most appropriate model for the given accuracy, memory, and energy constraints. Toward this, *we propose an algorithm to quantify the trade-off benefits of the SNN model candidates using our proposed reward function, and then select the one with the highest benefit.* The idea of our multi-objective reward function ($R$) is to prioritize the model that has high accuracy, small memory, and low energy consumption. The reward $R$ is defined as the resultant between the accuracy with the memory footprint and the energy consumption, as expressed in Equation 4.7. In this equation, $acc_x$ denotes the accuracy of the investigated SNN model ($x$). $m_{norm}$ denotes the normalized memory, which is defined as the ratio between the memory footprint of the investigated model ($mem_x$) and the floating-point model ($mem_{fp}$); see Equation 4.8. The memory footprint of the model is estimated by leveraging the number of weights ($N_{wgh}$) and the corresponding bitwidth ($BW_{wgh}$); see Equation 4.9. Meanwhile, $E_{norm}$ denotes the normalized energy consumption, which is defined as the ratio between the DRAM access energy of the approximate DRAM ($E_{DRAM\_approx}$) and the accurate one ($E_{DRAM\_accurate}$) for the investigated model; see Equation 4.10. To define the significance of memory and energy consumption with respect to the accuracy when calculating $R$, we employ $\mu$ and $\varepsilon$ as the adjustable trade-off variables for memory and energy consumption, respectively. Here, $\mu$ and $\varepsilon$ are the non-negative real numbers.

$$R(acc_x, m_{norm}, E_{norm}) = acc_x - (\mu \cdot m_{norm} + \varepsilon \cdot E_{norm}) \tag{4.7}$$

$$m_{norm} = \frac{mem_x}{mem_{fp}} \tag{4.8}$$

$$mem = N_{wgh} \cdot BW_{wgh} \tag{4.9}$$

$$E_{norm} = \frac{E_{DRAM\_approx}}{E_{DRAM\_accurate}} \tag{4.10}$$

### 4.4.6 Evaluation Methodology

Figure 4.37 shows the experimental setup and tools flow for evaluating our EnforceSNN framework, which are explained in the following.

**Accuracy Evaluation:** We employ PyTorch-based simulations [HSK+18] with 32-bit floating-point (FP32) and 8-bit fixed-point precision (i.e., FxP8 with "signed $Q1.6$" and "unsigned $Q1.7$" formats) that run on a multi-GPU machine, i.e., Nvidia GeForce RTX 2080 Ti. For network architecture, we consider the fully-connected network shown in Figure 2.12(a), with a different number of excitatory neurons, which are referred to as N-$i$ for conciseness with $i$ denoting the number of excitatory neurons. We use the rate coding converting data into spikes, and employ the MNIST and Fashion MNIST datasets. For comparison partners, we use the SNN model which is pre-trained without considering DRAM errors as the baseline. We perform an epoch of STDP-based unsupervised learning

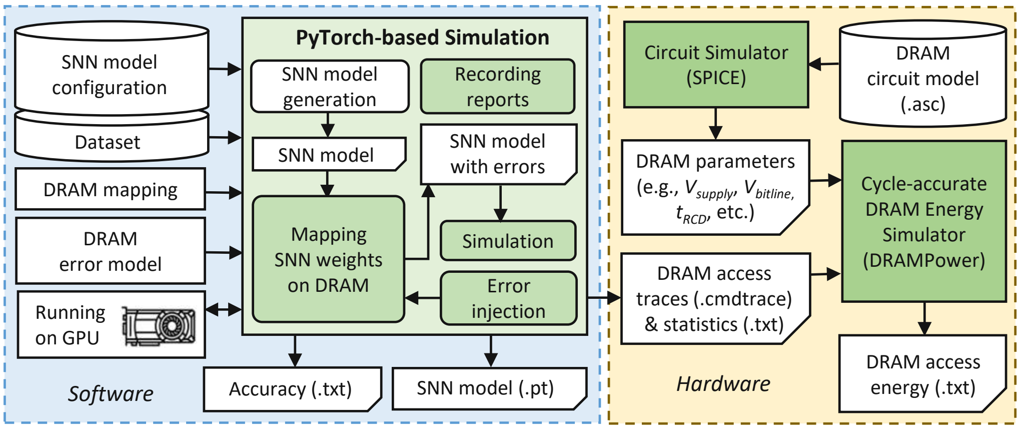

SNN model configuration
Dataset
DRAM mapping
DRAM error model
Running on GPU
Software
PyTorch-based Simulation
SNN model generation
Recording reports
SNN model
SNN model with errors
Mapping SNN weights on DRAM
Simulation
Error injection
Accuracy (.txt)
SNN model (.pt)
Circuit Simulator (SPICE)
DRAM circuit model (.asc)
DRAM parameters (e.g., V_supply, V_bitline, t_RCD, etc.)
Cycle-accurate DRAM Energy Simulator (DRAMPower)
DRAM access traces (.cmdtrace) & statistics (.txt)
DRAM access energy (.txt)
Hardware

Figure 4.37: Our experimental setup and tools flow.

through 60K experiments for each retraining process considering each combination of the SNN model, dataset, and training BER. Afterward, we perform inference through 10K experiments for each combination of the SNN model, dataset, and testing BER.

**DRAM Error Generation and Injection:** First, we generate bit errors based on the DRAM error model-0, and inject them into the DRAM cell locations, while considering the data mapping policy in DRAM. Afterward, the weight bits that are stored in the faulty DRAM cell locations (cells with errors), will be flipped. For the baseline data mapping, we place the weight bits in the subsequent address in a DRAM bank to maximize the DRAM burst feature, and if a DRAM bank is fully filled, then the weight bits are mapped in a different bank of the same DRAM chip. Meanwhile, we use the proposed DRAM mapping in Algorithm 7 for our EnforceSNN.

**DRAM Energy Evaluation:** We use the DRAM circuit model from [CYG+17a] and the SPICE simulator to extract the DRAM operational parameters (e.g., $V_{supply}$, $V_{bitline}$, $t_{RCD}$, $t_{RAS}$, $t_{RP}$), while considering the configuration of LPDDR3-1600 4Gb DRAM which is representative for the main memory of embedded systems. The accurate DRAM operates with 1.35V of $V_{supply}$, while the approximate one operates with 1.025V-1.325V of $V_{supply}$. Afterward, we use the state-of-the-art cycle-accurate DRAMPower [Cha14] that incorporates the DRAM access traces and statistics as well as the extracted DRAM parameters for estimating the DRAM access energy.

### 4.4.7 Results and Discussion

#### 4.4.7.1 Improvements of the SNN Error Tolerance

Figure 4.38 shows the accuracy of the baseline model and the EnforceSNN-improved model with accurate and approximate DRAM across different BER values, precision levels, i.e., FP32 and FxP8 (“signed $Q1.6$” and “unsigned $Q1.7$”), network sizes, and workloads (i.e., the MNIST and Fashion MNIST datasets). In general, we observe that the baseline model with approximate DRAM achieves lower accuracy than the baseline model with accurate DRAM, and the accuracy decreases as the BER increases. These trends are observed across different weight precision levels, network sizes, and datasets. The reason is that, the weights are changed (i.e., flipped) if they are stored in the faulty DRAM cells, and these weights are not trained to adapt to such bit flips. Therefore, the corresponding neuron behavior deteriorates from the expected behavior, hence decreasing accuracy. On the other hand, the EnforceSNN-improved model with approximate DRAM improves accuracy over the baseline model with accurate and approximate DRAM, across different BER values, network sizes, and datasets, as shown in ❶. We also observe that, the EnforceSNN-improved model with approximate DRAM improves the accuracy over the baseline model with accurate and approximate DRAM, even in the high error rate case (i.e., $BER = 10^{-2}$), as shown in ❷ for FP32 and ❸ for FxP8 weight precision levels. The reason is that, our EnforceSNN incorporates the error profiles from the approximate DRAM across different BER values in the training process, which makes the SNN model adaptive to the presence of DRAM errors, thereby improving the SNN error tolerance. For the MNIST dataset, a high error rate (i.e., $BER = 10^{-2}$) typically decreases the accuracy of the SNN-FxP8 more than the SNN-FP32, as shown in ❹. The reason is that, the MNIST dataset has a narrow weight distribution in each class to represent its digit features, hence bit errors may change the weight values significantly in the FxP8 precision than the FP32 due to its shorter bit-width. As a result, the corresponding neuron behavior deteriorates from its ideal behavior, hence degrading accuracy. For the Fashion MNIST dataset, the SNN-FxP8 may achieve higher accuracy than the SNN-FP32 in some cases, as shown in ❺. The potential reason is the following. The Fashion MNIST dataset has relatively more complex features than the MNIST dataset, hence having a wider weight distribution in each class to represent its various features which may overlap with features from other classes (i.e., non-unique features). Then, the quantization removes these non-unique features by eliminating the less significant bits of the trained weights (i.e., like the denoising effect), and the retraining makes the quantized weights adaptive to bit flips, thereby leading to higher accuracy than the non-quantized ones. Furthermore, we also observe that the accuracy of the SNN-FxP8 starts showing notable degradation at a high error rate (i.e., $BER = 10^{-2}$). For quantized models, in general, the "unsigned $Q1.7$" and "signed $Q1.6$" formats have similar trends and comparable accuracy as they represent similar weight values which differ only in the least significant fractional bit, thereby leading to similar neuron behavior and accuracy. These formats may have notable accuracy differences for some cases, such as after the retraining process, as shown by ❻. The possible reason is that, these formats have different bit positions for sign, integer, and fraction, thereby making the DRAM errors affect different weight bits and lead to different learning qualities during the respective fault-aware training.

In summary, our EnforceSNN maintains accuracy (i.e., no accuracy loss) as compared to the baseline with accurate DRAM when $BER \leq 10^{-3}$ across different datasets. Meanwhile, for higher BER values (i.e., $10^{-3} < BER \leq 10^{-2}$), our EnforceSNN still achieves higher accuracy than the baseline with accurate DRAM across different datasets. Therefore, these results show that *our EnforceSNN framework effectively improves the SNN error tolerance against DRAM errors with minimum retraining efforts.*

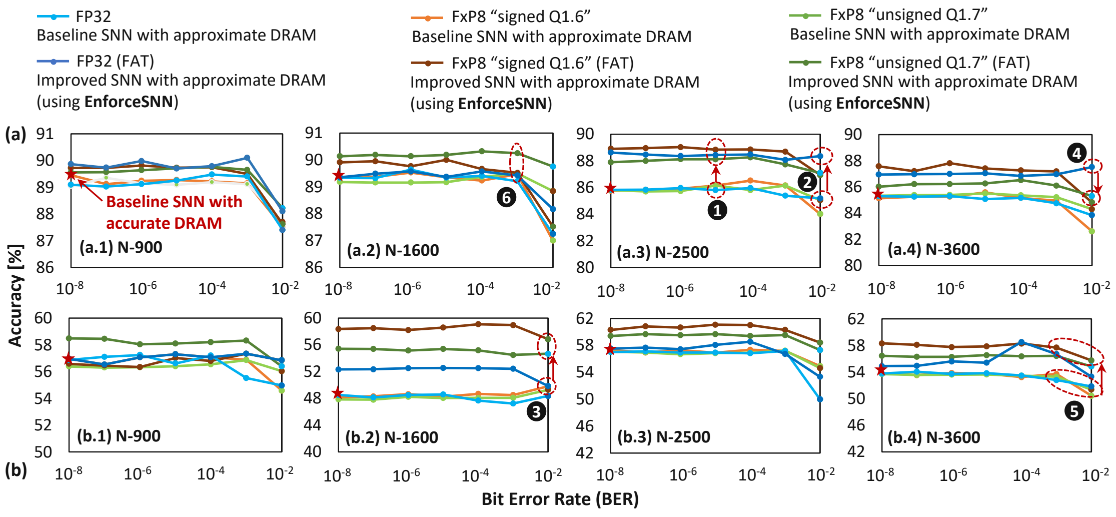


Figure 4.38: The accuracy of the baseline model with accurate and approximate DRAM, as well as the EnforceSNN-improved model with approximate DRAM for (a) MNIST and (b) Fashion MNIST datasets, across different precision levels, different BER values, and different network sizes.

#### 4.4.7.2 DRAM Access Energy Savings and Throughput Improvements

**DRAM Access Energy Savings:** Figure 4.39(a) shows the normalized energy consumption of the DRAM accesses for an inference (i.e., inferring one input sample) required by the baseline model and the EnforceSNN-improved model with accurate and approximate DRAM, across different $V_{supply}$ values, precision levels, network sizes, and workloads. We observe that different network sizes show similar normalized DRAM access energy, hence we only show a single figure representing the experimental results for all network sizes. For accurate DRAM cases across different network sizes, the baseline model achieves 75% DRAM energy saving when it employs the quantization technique, while our EnforceSNN-improved model achieves 75.1% DRAM energy saving due to the quantization and the proposed DRAM mapping policy, as shown in ❼. Meanwhile, the difference in these DRAM energy savings comes from the DRAM mapping policy. That is, our EnforceSNN optimizes the DRAM energy-per-access by maximizing the row buffer hits and the multi-bank burst feature, thereby having fewer row buffer conflicts than the baseline which only exploits the single-bank burst feature. For the FP32 precision across different network sizes, employing the approximate DRAM in the baseline model reduces the DRAM energy savings by up to 39.2% as compared to employing the accurate DRAM. Meanwhile, employing the approximate DRAM in the EnforceSNN-improved model reduces the DRAM energy savings by up to 39.5% as compared to employing the accurate DRAM, as shown in ❽. These energy savings come from the reduced DRAM energy-per-access due to the reduction of operational $V_{supply}$. Moreover, the difference in energy savings between the baseline and our EnforceSNN also comes from the DRAM mapping policy. For the FxP8 precision (i.e., "signed $Q1.6$" and "unsigned $Q1.7$") across

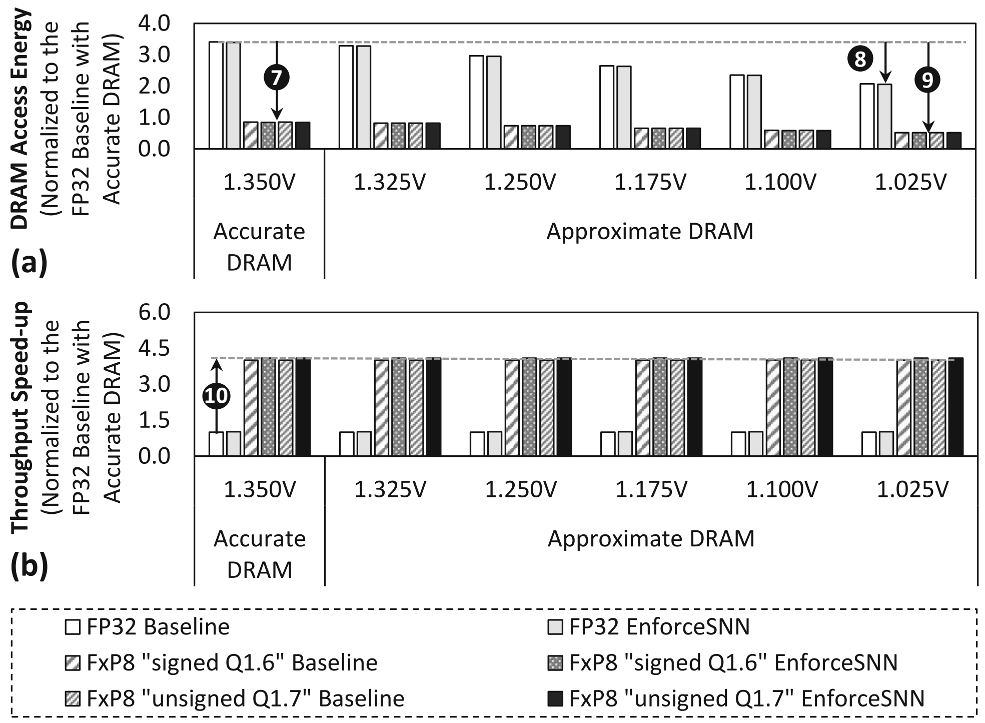


Figure 4.39: (a) The normalized DRAM access energy for an inference incurred by the baseline model and the EnforceSNN-improved model with accurate and approximate DRAM, and (b) the normalized speed-up of DRAM data throughput for an inference achieved by our EnforceSNN-improved model over the baseline model, across different $V_{supply}$ values, different workloads (datasets), and different network sizes (N-900, N-1600, N-2500, and N-3600). These results are applicable to all network sizes. They are also applicable for both the MNIST and Fashion MNIST datasets, as these workloads have similar DRAM access energy, due to the same number of weights and number of DRAM accesses for an inference.

different network sizes, employing the approximate DRAM in the baseline model reduces the DRAM energy savings by up to 84.8% over employing the accurate one. Meanwhile, employing the approximate DRAM in the EnforceSNN-improved model reduces the DRAM energy savings by up to 84.9% over employing the accurate one, as shown in ❾. These energy savings come from the reduced weight precision and the reduced DRAM energy-per-access due to $V_{supply}$ reduction. Moreover, the difference in energy savings between the baseline and our EnforceSNN also comes from the DRAM mapping policy.

**DRAM Throughput:** We observe that our EnforceSNN-improved model obtains 4.1x throughput speed-up over the baseline model across different $V_{supply}$ values, workloads, and network sizes; see ❿ in Figure 4.39(b). It is achieved through (1) the quantization technique which reduces the number of DRAM accesses, and (2) our proposed DRAM mapping policy which optimizes the DRAM latency-per-access by maximizing the row buffer hits and the multi-bank burst features. The results also show that the "unsigned $Q1.7$" and "signed $Q1.6$" achieve comparable DRAM access energy savings and throughput improvements since they employ the same bitwidth of weights, thereby having similar DRAM access behavior.

In summary, the results in Figure 4.39 indicate that *our EnforceSNN framework substantially reduces the DRAM access energy by employing the reduced-voltage approximate DRAM and our efficient DRAM mapping policy, while effectively improving the DRAM data throughput mainly due to the quantization.*

#### 4.4.7.3 Model Selection under Design Trade-Offs

Figure 4.40 and Figure 4.41 show the results of the accuracy-memory-energy trade-offs for the MNIST and Fashion MNIST datasets, respectively. In this evaluation, the quantized models consider the FxP8 precision with "signed $Q1.6$" format. For the given SNN model candidates, we observe that the models that incur small memory typically employ FxP8 precision, as shown in Figure 4.40(a) for the MNIST and Figure 4.41(a) for the Fashion MNIST. Considering that the accuracy of the FxP8-based models is comparable to the FP32-based models, we narrow down the candidates to only the FxP8-based models.

To analyze the design trade-offs, we explore the impact of different $\mu$ and $\varepsilon$ values on the rewards. For instance, if we consider that the accuracy should have a higher priority than the memory and the energy consumption, we set $\mu$ and $\varepsilon$ low (e.g., $\mu = 0$ and $\varepsilon = 0$). Meanwhile, if we consider that the memory should have a higher priority than the energy consumption, we set $\mu$ higher than $\varepsilon$ (e.g., $\mu = 10$ and $\varepsilon = 0$). For both cases, the highest reward is achieved by the EnforceSNN-improved N-1600 FxP8 for the MNIST and the EnforceSNN-improved N-2500 FxP8 for the Fashion MNIST under $10^{-5}$ error rate; see Ⓓ for $\mu = 0$ and $\varepsilon = 0$ case, and see Ⓔ for $\mu = 10$ and $\varepsilon = 0$ case. The reason is that, these models employ our efficient FAT technique to improve their error tolerance, thereby leading to high accuracy under a high error rate. We also observe that having $\mu$ higher than $\varepsilon$ makes the high rewards shift towards smaller models, as shown by Ⓔ in Figure 4.40(c) and Figure 4.41(c). The reason is that a higher $\mu$ makes the small $m_{norm}$ have a smaller impact on the reward reduction than the large $m_{norm}$, thereby maintaining the high reward values. If the energy consumption should have a higher priority than the memory footprint, we set $\mu$ lower than $\varepsilon$ (e.g., $\mu = 0$ and $\varepsilon = 10$). The highest reward is achieved by the EnforceSNN-improved N-1600 FxP8 under $10^{-5}$ error rate for the MNIST and the EnforceSNN-improved N-2500 FxP8 under $10^{-4}$ error rate for the Fashion MNIST. In this case, we observe that high rewards are shifted towards models with smaller energy consumption (represented by higher BER); see Ⓕ in Figure 4.40(d) and Figure 4.41(d). The reason is that a higher $\varepsilon$ makes the small $E_{norm}$ have a smaller impact on the reward reduction than the large $E_{norm}$, thereby maintaining the high reward values. Furthermore, if the memory and the energy consumption should have a higher priority than the accuracy, we set $\mu$ and $\varepsilon$ high (e.g., $\mu = 10$ and $\varepsilon = 10$). The highest reward is achieved by the EnforceSNN-improved N-1600 FxP8 under $10^{-5}$ error rate for the MNIST and the EnforceSNN-improved N-2500 FxP8 under $10^{-4}$ error rate for

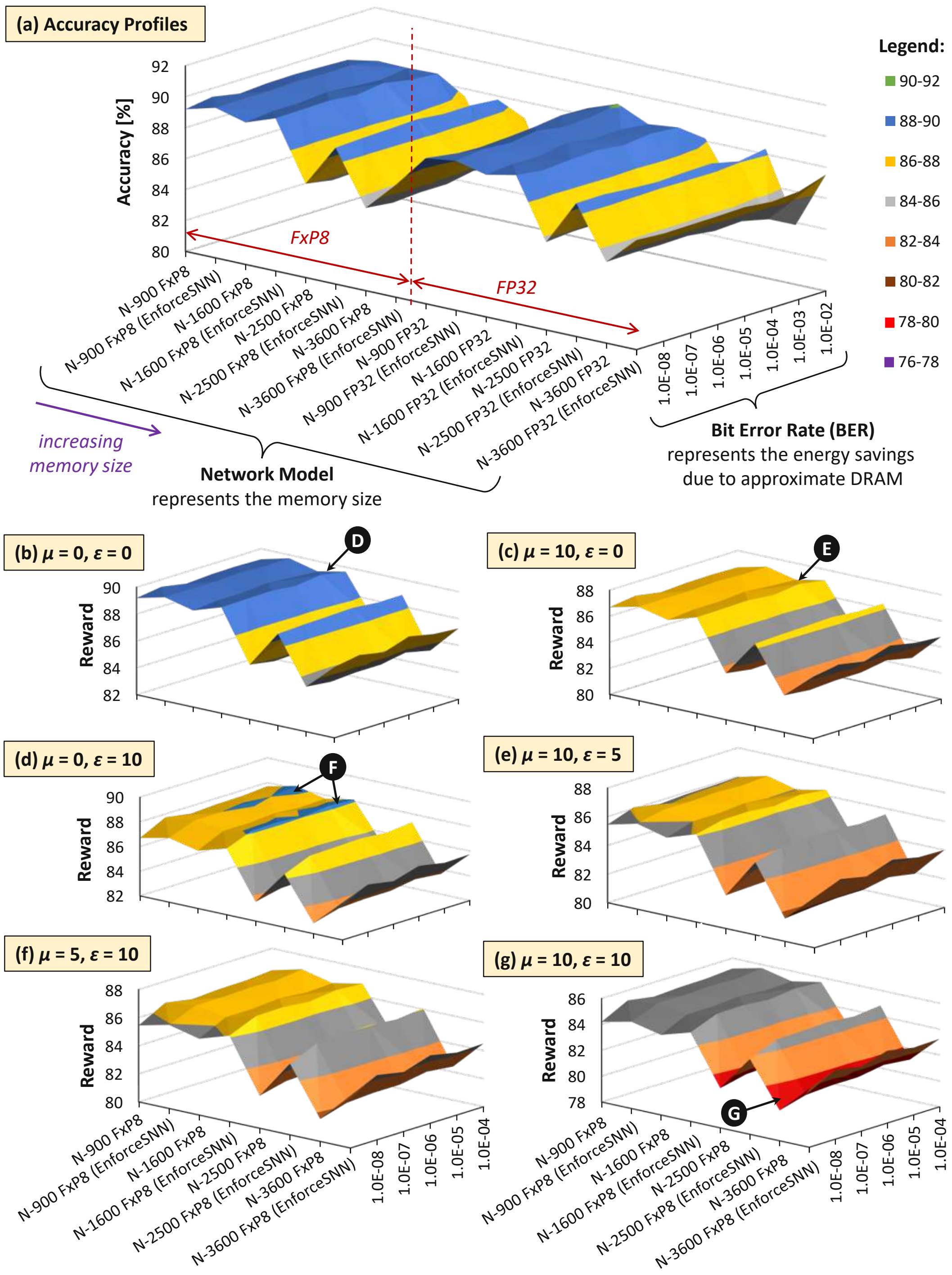


Figure 4.40: The trade-offs among accuracy, memory footprint, and energy consumption for the MNIST dataset. (a) Accuracy profiles of SNN models. (b-g) Reward profiles of SNN models. The network sizes represent the memory sizes, and the BER values represent the energy savings from approximate DRAM.

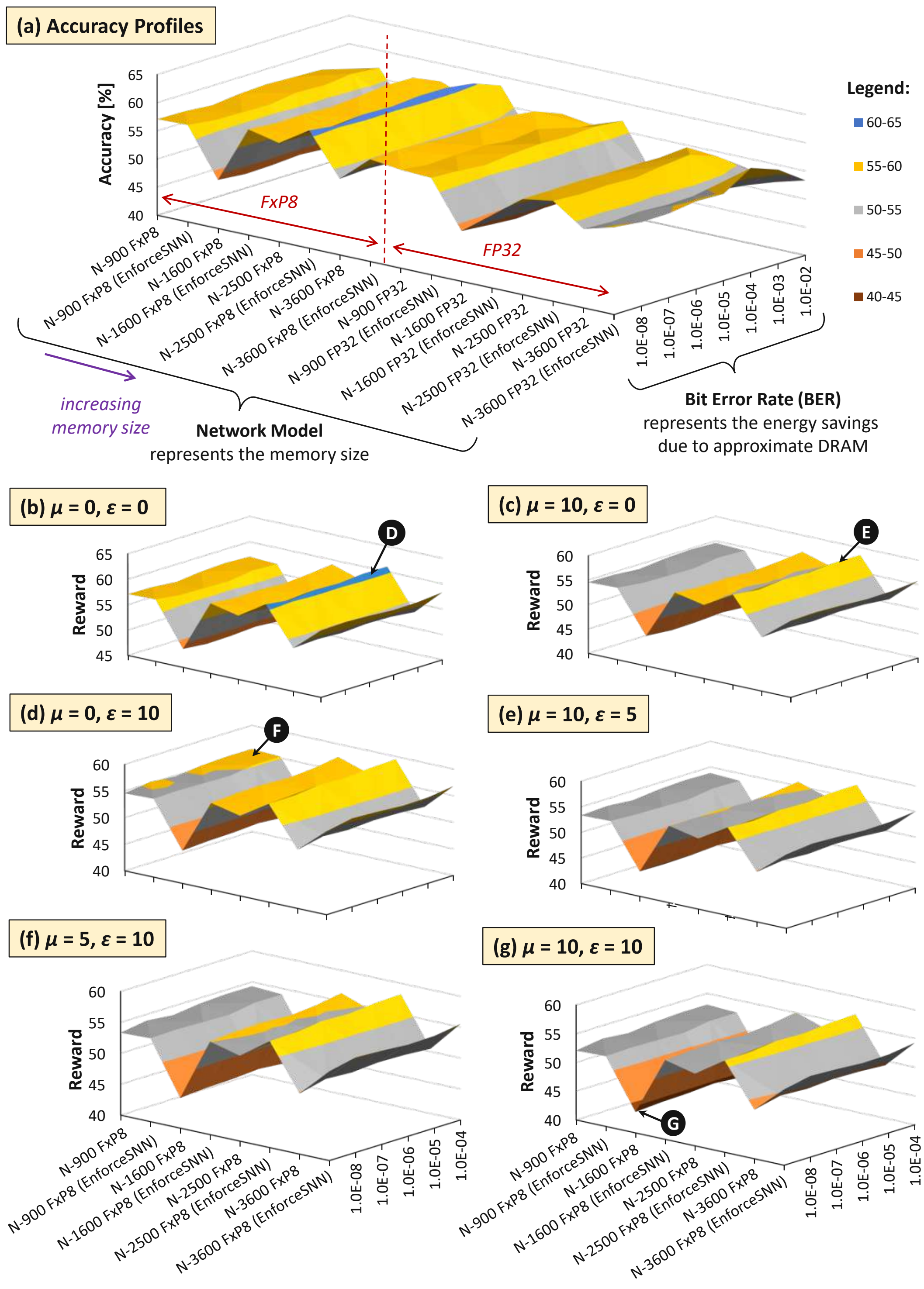


Figure 4.41: The trade-offs among accuracy, memory, and energy consumption for the Fashion MNIST dataset. (a) Accuracy profiles of SNN models. (b-g) Reward profiles of SNN models. The network sizes represent the memory sizes, and the BER values represent the energy savings from approximate DRAM.

the Fashion MNIST. In this case, high rewards are shifted towards models with smaller memory and energy consumption (represented by high BER), but their overall rewards decrease as the values of $\mu$ and $\varepsilon$ increase; see G in Figure 4.40(e) and Figure 4.41(e). The reason is that, higher $\mu$ and $\varepsilon$ jointly make the $m_{norm}$ and $E_{norm}$ decrease the reward. It means that if we want to significantly reduce the memory footprint and energy consumption, we have to accept more accuracy degradation.

In summary, the results in Figure 4.40 and Figure. 4.41 show that *our EnforceSNN framework has an effective algorithm to trade-off the accuracy, memory footprint, and energy consumption of the given SNN models*, thereby providing good applicability for diverse embedded applications with their respective constraints.

#### 4.4.7.4 Optimization of the Retraining Costs

The conventional FAT for neural networks usually injects errors at an incremental rate during the retraining process from the minimum value to the maximum one for avoiding accuracy collapse [KOY+19]. Therefore, in this work, the conventional FAT considers $BER = \{10^{-8}, 10^{-7}, 10^{-6}, ..., 10^{-2}\}$, while our efficient FAT in EnforceSNN only considers $BER = \{10^{-4}, 10^{-3}, 10^{-2}\}$ in the retraining process.

**Retraining Speed-ups:** The conventional FAT with FxP8 (cFAT8) obtains speed-up over the one with FP32 (cFAT32) by up to 1.16x and 1.14x for the MNIST and the Fashion MNIST respectively, since the cFAT8 employs quantized weights, thereby leading to a faster error injection and learning process. Meanwhile, our efficient FAT with FP32 (eFAT32) obtains a 2.33x speed-up over the cFAT32, since our eFAT32 has fewer iterations of the retraining process. Furthermore, we also observe that our efficient FAT with FxP8 (eFAT8) obtains more speed-up over the cFAT32, i.e., by up to 2.71x for the MNIST and 2.65x for the Fashion MNIST as shown by H in Figure 4.42(a), since our eFAT8 employs quantized weights in addition to fewer iterations of the retraining process.

**Retraining Energy Savings:** The cFAT8 achieves energy saving over the cFAT32 by up to 13.9% for the MNIST and 12% for the Fashion MNIST, since the cFAT8 employs quantized weights which incur lower energy consumption during the error injection and learning process. Meanwhile, our eFAT32 achieves energy saving over the cFAT32 by 57.1%, as the eFAT32 performs fewer iterations of the retraining process as compared to the cFAT32. Our eFAT8 achieves further energy saving over the cFAT32, i.e., by up to 63.1% for the MNIST and by up to 62.3% for the Fashion MNIST as shown by I in Figure 4.42(b), since it employs quantized weights in addition to fewer iterations of the retraining process, thereby leading to a higher energy saving.

**Carbon Emission Reduction:** The retraining process also poses additional challenges that correspond to environmental concerns, i.e., carbon emission. Recent works have highlighted that the carbon emission from neural network training should be minimized to prevent the increasing rates of natural disasters [SGM19, SGM20]. To estimate the carbon emission of neural network training, the work of [SGM19] proposed Equation 4.11 and Equation 4.12. In these equations, $CO_2e$ denotes the estimated carbon ($CO_2$)

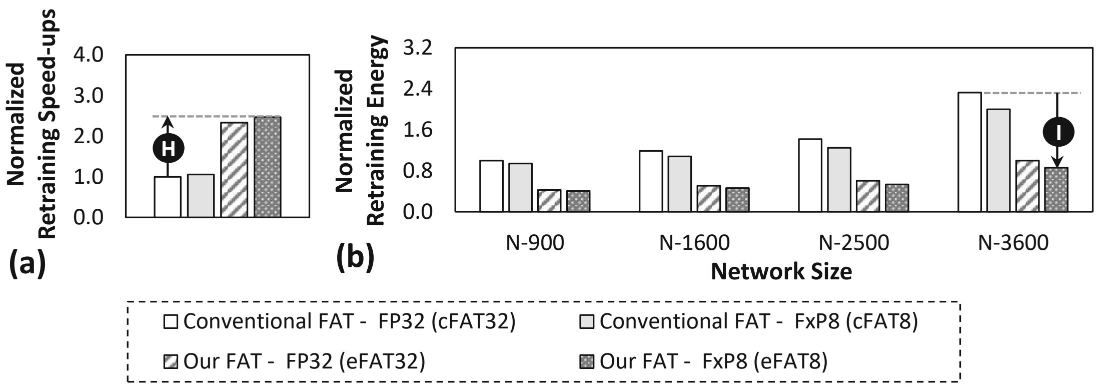


Figure 4.42: (a) The retraining speed-ups across different network sizes (i.e., N-900, N-1600, N-2500, and N-3600), and (b) the retraining energy for the MNIST, which are normalized to the conventional FAT with FP32 for a 900-neuron network. The results for the Fashion MNIST show similar trends to the MNIST since these workloads have the same number of weights and number of DRAM accesses for a training phase. Here, FxP8 represents both the "signed $Q1.6$" and "unsigned $Q1.7$" formats.

emission during the training, which is a function of the total power during the training ($p_t$). Meanwhile, $t$ is the training duration, $p_c$ is the average power from all CPUs, $p_r$ is the average power from all main memories (DRAMs), $p_g$ is the average power from a GPU and $g$ is the number of GPUs.

$$CO_2e = 0.954 \cdot p_t \tag{4.11}$$

$$p_t = \frac{1.58 \cdot t(p_c + p_r + g \cdot p_g)}{1000} \tag{4.12}$$

These equations indicate that if we assume $p_c$, $p_r$, $p_g$, and $g$ are the same for different FAT techniques, then the difference will come from the training duration $t$. Therefore, our efficient FAT in EnforceSNN employs fewer iterations of the retraining process than the conventional FAT, thereby producing less carbon emission. Moreover, our EnforceSNN also reduces the operational power of the main memory (DRAM) through the reduced-voltage approximation approach, thereby further reducing the emission.

In summary, *our EnforceSNN framework effectively offers speed-up of retraining time, reduction of energy consumption for retraining, and less carbon emission than the conventional FAT technique*, thereby making it more friendly for our environments.

### 4.4.7.5 Further Discussion

Previous works that exploit the reduced-voltage DRAM concept mainly aim at improving the energy efficiency of mobile systems [HYAK+20], personal computing systems [NLSU+21, FFW22], and server systems [NLSU+21, DFG+11, DMR+11, DMB+12a, DMB+12b]. This concept is also employed for minimizing the energy consumption of deep neural networks (DNNs) [KOY+19]. Since SNNs have different data representations, computation models, and learning rules as compared to DNNs, our EnforceSNN provides a different framework with different techniques that are crafted specifically for improving the resilience and energy efficiency of SNNs. Furthermore, the reduced-voltage DRAM is also beneficial for generating noise (i.e., from DRAM errors) for obfuscating the intellectual property (IP) against security threats, such as IP stealing [XTAQ20].

Our EnforceSNN framework can be put in the approximate computing field, especially in the context of the approximation for main memory through voltage scaling [VCRR15, XMK16, Mit16]. Furthermore, some of the techniques in our EnforceSNN framework are suitable for different domains outside SNNs: (1) DRAM voltage reduction for optimizing the DRAM access energy, (2) quantization for reducing the memory footprint, and (3) error-aware DRAM data mapping policy for minimizing the negative impact of DRAM errors to the data. These techniques are applicable for error-tolerant applications, such as image/video processing (e.g., data compression) and data analytic applications (e.g., data clustering).

### 4.4.8 Summary of EnforceSNN Framework

We propose a novel EnforceSNN framework to achieve a resilient and energy-efficient SNN inference considering reduced-voltage-based approximate DRAM, through weight quantization, error-aware DRAM mapping, SNN error-tolerance analysis, efficient error-aware SNN training, and effective SNN model selection. Our EnforceSNN achieves no accuracy loss for BER $\leq 10^{-3}$ with minimum retraining costs as compared to the baseline SNN with accurate DRAM while achieving up to 84.9% of DRAM energy saving and up to 4.1x speed-up of DRAM data throughput. In this manner, our work may enable efficient SNN inference for energy-constrained embedded systems like Edge-AI devices.

## 4.5 SpikeDyn: A Framework for SNNs with Unsupervised Continual Learning

This section aims at addressing **Problem-4** with the solution for enabling a lightweight and energy-efficient unsupervised continual learning mechanism in SNNs for adapting to dynamic environments.

### 4.5.1 Motivational Study

To understand the characteristics (i.e., accuracy, memory, and energy consumption) of SNNs under dynamic environment settings, we perform an experimental case study. Here, we consider an unsupervised learning-based SNN model that is widely considered in the SNN community, i.e., the model with fully-connected architecture and a pair of excitatory and inhibitory layers, that has been employed in state-of-the-art works [PARR18, AR20]; see Figure 2.12(a). In this case study, we provide dynamic environments by feeding consecutive task (i.e., class) changes to the network. If we consider the MNIST dataset,

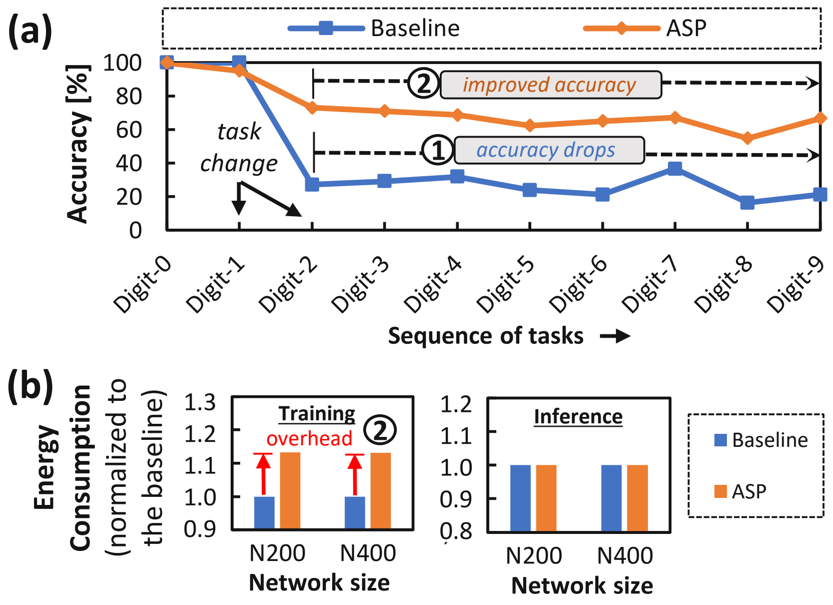


Figure 4.43: (a) Per-digit accuracy for a 400-neuron SNN. (b) Energy consumption for networks with 200 excitatory neurons (N200) and 400 excitatory neurons (N400). Here, the baseline refers to the pair-based weight-dependence STDP [DC15], while the ASP refers to the adaptive synaptic plasticity [PARR18].

then first, a stream of samples for digit-0 is fed to the network. Afterward, the task is changed to digit-1. This process is repeated for other tasks without re-feeding previous tasks, and each task has the same number of samples. More details of the experimental setup are presented in Section 4.5.5. Our experimental results are presented in Figure 4.43, from which we make the following key observations.

① The baseline does not efficiently learn new tasks from digit-2 onward, as most of the synapses are already occupied by previously learned tasks (digit-0 and digit-1), and mix new information with the existing ones.

② The state-of-the-art work improves the accuracy over the baseline at a cost of an energy overhead due to: (a) a large number of weights and neuron parameters from excitatory and inhibitory layers, and (b) complex exponential calculations for computing multiple spike traces, membrane- and threshold-potential decay, and weight decay.

### 4.5.2 Scientific Research Challenges

Our above observations expose several challenges that need to be solved to address the targeted problem as the following.

- The SNN systems should employ a simple yet effective learning algorithm at run time that achieves high accuracy, in both dynamic and non-dynamic environment settings.

- It should reduce the non-significant weights or neuron parameters, and the complex exponential calculations, to optimize the energy consumption.
- The memory and energy constraints should be considered in the design process to meet the design specification.

***Required:*** *A lightweight technique that effectively performs unsupervised continual learning for SNNs in a memory-efficient and energy-efficient manner under dynamic environments is required.*

### 4.5.3 Novel Contributions

To address the above challenges, we propose *SpikeDyn, a novel framework for developing energy-efficient Spiking Neural Networks with unsupervised continual learning capabilities in Dynamic environments.* The SpikeDyn employs the following key mechanisms (see an overview in Figure 4.44).

1. **Reducing the energy consumption of the neuronal operations** by replacing the inhibitory neurons with the direct lateral inhibitory connections.
2. **An SNN model search algorithm** under the given memory- and energy-constraints. It quickly estimates the memory footprint and energy consumption of the investigated SNN models using our analytical models that leverage the network parameters, the bit precision, the energy for processing an input sample, and the number of samples.
3. **An algorithm for enabling an efficient unsupervised continual learning** that leverages (a) adaptive learning rates, (b) synaptic weight decay, (c) adaptive membrane threshold potential, and (d) reduction of the spurious weight updates.

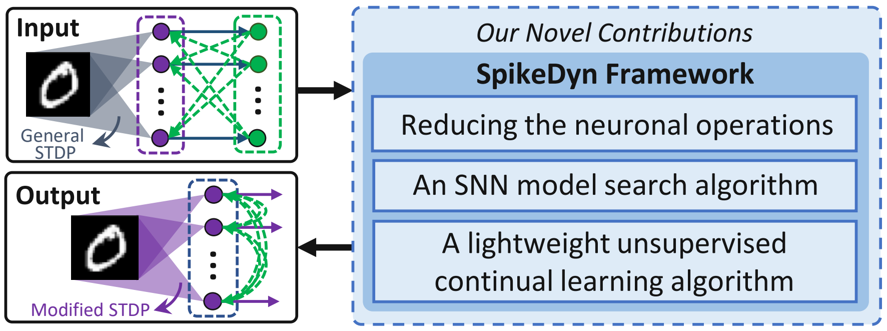

Input
General STDP
Output
Modified STDP
Our Novel Contributions
SpikeDyn Framework
Reducing the neuronal operations
An SNN model search algorithm
A lightweight unsupervised continual learning algorithm

Figure 4.44: The overview of novel contributions (shown in the blue boxes).

### 4.5.4 SpikeDyn Framework

The SpikeDyn framework employs the following key mechanisms for enabling lightweight unsupervised continual learning for SNNs (the overview is shown in Figure 4.45), which are explained in the subsequent sections.

1. **Reducing the operations in SNN model (Section 4.5.4.1)** to minimize the energy consumption by replacing the inhibitory neurons with the direct lateral inhibitions. Hence, the operations in the inhibitory layer of the SNN model that supports unsupervised continual learning settings, are eliminated.
2. **An SNN model search algorithm (Section 4.5.4.2)** that explores a different number of excitatory neurons to find SNN model size that meet the memory and energy constraints. To quickly perform the search, we also propose analytical models that incorporate the following.
   - Number of weights and neuron parameters, as well as bit precision: They are used for estimating the memory footprint;
   - Energy consumption for processing an input sample, and the number of samples that will be processed: They are used for estimating the energy consumption.
3. **An unsupervised continual learning (Section 4.5.4.3)** that employs the following.
   - *Adaptive learning rates* that determine the potentiation and depression factors in the STDP-based learning, using the presynaptic and postsynaptic spike activities.
   - *Synaptic weight decay* that helps the network to gradually remove the weak synaptic connections (which represent old and insignificant information), thereby enabling the synapses to learn new information.
   - *Adaptive membrane threshold potential* that provides balance in the neurons' internal, so that the neuron generates spikes only when the corresponding synapses need to learn the input features. It is determined by the neurons' threshold potential value and its decay rate.
   - *Reduction of the spurious weight updates* considering the presynaptic and post-synaptic spike event by employing timestep-based updates to carefully perform weight potentiation and depression.

#### 4.5.4.1 Reducing the Neuronal Operations

State-of-the-art works in the unsupervised continual learning for SNNs, employ the network architecture shown in Figure 2.12(a), which consists of the input, excitatory, and inhibitory layers [PARR18, AR16]. We observe that the inhibitory neurons have different parameters from the excitatory ones to be saved in memory. Therefore, employing such an architecture will consume high memory and energy. To address this issue, we reduce the neuron operations in the inhibitory layer to substantially decrease the memory footprint and energy consumption, as shown in Figure 4.46(a)-(c). We also observe that, the optimized architecture still achieves a similar accuracy profile as of the baseline, as shown in Figure 4.46(d). Therefore, we will improve the accuracy of the optimized architecture with our learning mechanism, as discussed in Section 4.5.4.3.

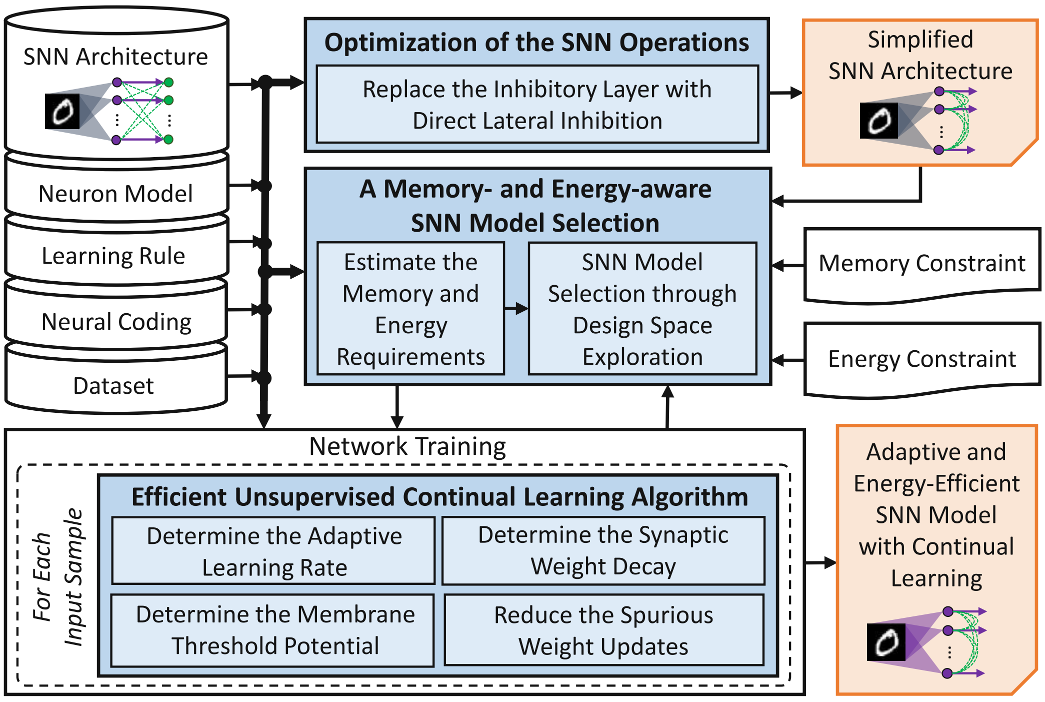

SNN Architecture
Neuron Model
Learning Rule
Neural Coding
Dataset
Optimization of the SNN Operations
Replace the Inhibitory Layer with Direct Lateral Inhibition
Simplified SNN Architecture
A Memory- and Energy-aware SNN Model Selection
Estimate the Memory and Energy Requirements
SNN Model Selection through Design Space Exploration
Memory Constraint
Energy Constraint
Network Training
For Each Input Sample
Efficient Unsupervised Continual Learning Algorithm
Determine the Adaptive Learning Rate
Determine the Synaptic Weight Decay
Determine the Membrane Threshold Potential
Reduce the Spurious Weight Updates
Adaptive and Energy-Efficient SNN Model with Continual Learning

Figure 4.45: An overview of the SpikeDyn framework. The novel contributions are highlighted in blue.

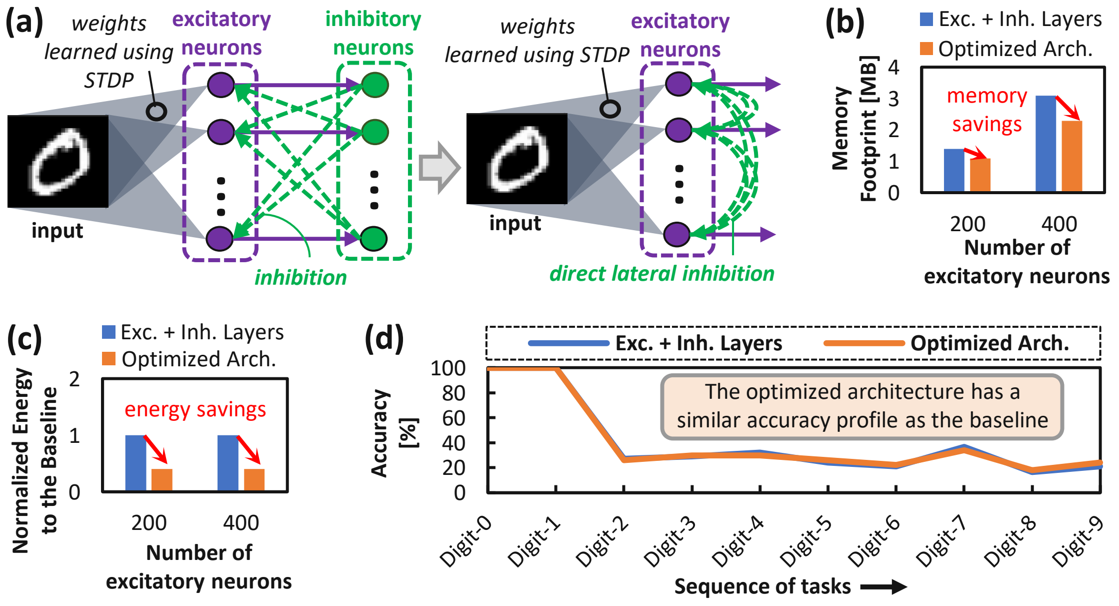


Figure 4.46: (a) Replacing the inhibitory neurons with the direct lateral inhibitions. The optimized architecture reduces (b) memory and (c) energy, but still has a similar (d) training accuracy profile in dynamic scenarios like the baseline architecture.

### 4.5.4.2 An SNN Model Search Algorithm

Each application use-case typically has memory- and energy-constraints that need to be considered in the model generation. Toward this, we propose an algorithm to search for an appropriate model size for a given SNN architecture that meets the design constraints (see Algorithm 9). The idea is to explore different sizes of an SNN model and estimate their memory and energy consumption in both training and inference phases, using the proposed analytical models as discussed in the following.

**Algorithm 9** Pseudo-code of the proposed search algorithm

```
INPUT: (1) Memory constraint ($mem_c$); (2) Energy constraints: training ($E_{ct}$), inference
    ($E_{ci}$); (3) SNN model ($model$): number of neurons ($model.n_{exc}$), size ($model.mem$), energy
    of training ($model.E_t$) and inference ($model.E_i$); (4) Energy for one sample: training ($E_{1t}$),
    inference ($E_{1i}$); (5) number of additional neurons ($n_{add}$);
OUTPUT: SNN model ($model$);
BEGIN
    Initialization:
 1: $model.n_{exc} = 0$;
 2: $model.mem = 0$;
    Process:
 3: while $model.mem \leq mem_c$ do
 4:   if ($model.n_{exc} > 0$) then
 5:     perform training with 1 sample using Algorithm 10;
 6:     calculate $E_{1t}$; // for 1 sample
 7:     estimate $model.E_t$; // for all samples
 8:     if ($model.E_t \leq E_{ct}$) then
 9:       perform inference with 1 sample;
10:       calculate $E_{1i}$; // for 1 sample
11:       estimate $model.E_i$; // for all samples
12:       if ($model.E_i \leq E_{ci}$) then
13:         save $model$;
14:   $model.n_{exc} += n_{add}$;
15:   estimate $model.mem$;
16: return $model$;
END
```

For each investigated SNN model size, the memory footprint ($mem$) is estimated using $mem = (P_w + P_n) \cdot BP$, which leverages the number of weights ($P_w$) and neuron parameters ($P_n$), as well as the bit precision ($BP$). The reason is that, the above aspects are dominant factors in determining the size of an SNN model. Meanwhile, the total energy consumption ($E$) is estimated using $E = E_1 \cdot N$, which leverages the energy for processing a single sample ($E_1$), and the number of samples that will be processed ($N$). The number of samples $N$ is important, as the deployed systems might have a different number of samples available from the environment. If the estimated memory $mem$ and energy $E$ are within the memory constraint ($mem_c$) and energy constraint ($E_c$) respectively, then the investigated SNN model is selected as the candidate of solution. Our algorithm then selects the largest-sized SNN model from the candidates as the solution, since a larger network usually can achieve higher accuracy [PS20]. We validate our analytical models against the actual execution runs, see the results presented in Figure 4.47(a) for memory footprint, and Figure 4.47(b)-(c) for energy consumption of training and inference, respectively. The results show that our analytical models achieve less than 5% errors compared to the actual runs. Thus, they are suitable for a fast estimation. Employing our algorithm is beneficial, rather than actually running all possible SNN configurations and selecting the desired one at the end, since it saves the exploration time, as shown in Figure 4.47(d)-(e).

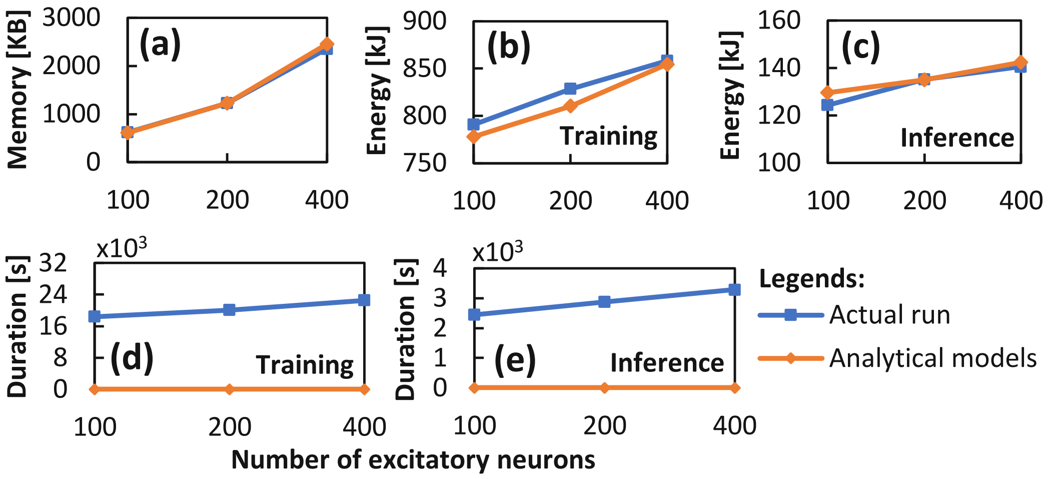


Figure 4.47: Validation of our analytical models against the actual runs in terms of (a) the memory footprint, and the energy consumption for (b) training and (c) inference, using the full MNIST dataset. Our algorithm also reduces the exploration time over the actual runs for both (d) training and (e) inference.

### 4.5.4.3 Proposed Learning Algorithm

Our SpikeDyn employs an algorithm that incorporates the following techniques (the pseudo-code is presented in Algorithm 10).

**Adaptive Learning Rates:** Our algorithm employs the potentiation factor ($k_p$) and the depression factor ($k_d$) to adaptively determine the learning rates for weight potentiation and depression. The idea is to adjust the potentiation factor $k_p$ to have a high value when the corresponding synapses need to learn input features, which is indicated by the occurrences of postsynaptic spikes. The value of $k_p$ is obtained by normalizing the maximum accumulated postsynaptic spikes ($maxSp_{post}$) with the spike threshold ($Sp_{th}$); see Equation 4.13. Meanwhile, the depression factor $k_d$ provides weight depression when the corresponding synapses need to weaken the connections, which is indicated by no occurrences of postsynaptic spikes. The value of $k_d$ is obtained by the ratio between the maximum accumulated postsynaptic spikes ($maxSp_{post}$) and presynaptic spikes ($maxSp_{pre}$); see Equation 4.13. These factors are incorporated into the improved

**Algorithm 10** Pseudo-code of the proposed learning algorithm

```
INPUT: (1) Simulation time for an input ($t_{sim}$);
    (2) Timestep ($t_{step}$);
    (3) SNN parameters: # of neurons ($n_{exc}$), # of synapses-per-neuron ($n_{syn}$), accumulated
    presynaptic spikes ($N_{sp\_pre}$) and accumulated postsynaptic spikes ($N_{sp\_post}$);
    (4) Presynaptic spike ($sp_{pre}$), postsynaptic spike ($sp_{post}$);
OUTPUT: Synaptic weight update ($\Delta w$);
BEGIN
    Initialization:
 1: $\Delta w[n_{exc}, n_{syn}] = zeros[n_{exc}, n_{syn}]$;
 2: $N_{sp\_pre}[n_{exc}, n_{syn}] = zeros[n_{exc}, n_{syn}]$;
 3: $N_{sp\_post}[n_{exc}] = zeros[n_{exc}]$;
    Process:
 4: for ($t = 0$ to $(t_{sim} - 1)$) do
 5:     for ($i = 0$ to $(n_{exc} - 1)$) do
 6:         for ($j = 0$ to $(n_{syn} - 1)$) do
 7:             if $sp_{pre}$ then
 8:                 $N_{sp\_pre}[i, j]$ += 1;
 9:             if $sp_{post}$ then
10:                 $N_{sp\_post}[i]$ += 1;
11:     if (($t \mod t_{step}$) == 0) then
12:         $maxSp_{pre} = max(N_{sp\_pre})$;
13:         $maxSp_{post} = max(N_{sp\_post})$;
14:         if (no $sp_{post}$ within $t_{step}$) then
15:             update $\Delta w[:,:]$ using Equation 4.14; // weight depression
16:         else
17:             $m \leftarrow index(max(N_{sp\_post}))$;
18:             update $\Delta w[m,:]$ using Equation 4.14; // weight potentiation
19:     else
20:         update $\Delta w[:,:]$ using weight decay;
21:     return $\Delta w$;
END
```

STDP-based learning algorithm; see Equation 4.14. Here, $\Delta w$ denotes the weight change, $\eta_{pre}$ and $\eta_{post}$ denote the learning rate for a presynaptic and postsynaptic spike, $x_{pre}$ and $x_{post}$ denote the presynaptic and postsynaptic trace, respectively.

$$k_p = \left\lceil \frac{maxSp_{post}}{Sp_{th}} \right\rceil \quad \text{and} \quad k_d = \frac{maxSp_{post}}{maxSp_{pre}} \tag{4.13}$$

$$\Delta w = \begin{cases} -k_d \cdot \eta_{pre} \cdot x_{post} & \text{on depression update time} \\ k_p \cdot \eta_{post} \cdot x_{pre} & \text{on potentiation update time} \end{cases} \tag{4.14}$$

**Synaptic Weight Decay:** We employ a weight decay to gradually remove the old and insignificant information, which is represented by small weight values. It follows Equation 4.15, with $\tau_{decay}$ denotes the decay time constant and $w_{decay}$ denotes the weight decay rate.

$$\tau_{decay} \cdot \frac{dw}{dt} = -w_{decay} \cdot w \tag{4.15}$$

In this manner, weak connections will get more disconnected over the training period. We define the value of $w_{decay}$ to be inversely proportional to the size of the network ($w_{decay} \propto 1/n_{exc}$), with $n_{exc}$ denotes the number of excitatory neurons. The reason is that, a smaller network has less number of synapses for learning new information, while retaining the old ones. Therefore, it needs to forget the old information at a faster rate than the larger network. We observe that an appropriate $w_{decay}$ can improve accuracy, as shown by the label-① in Figure 4.48.

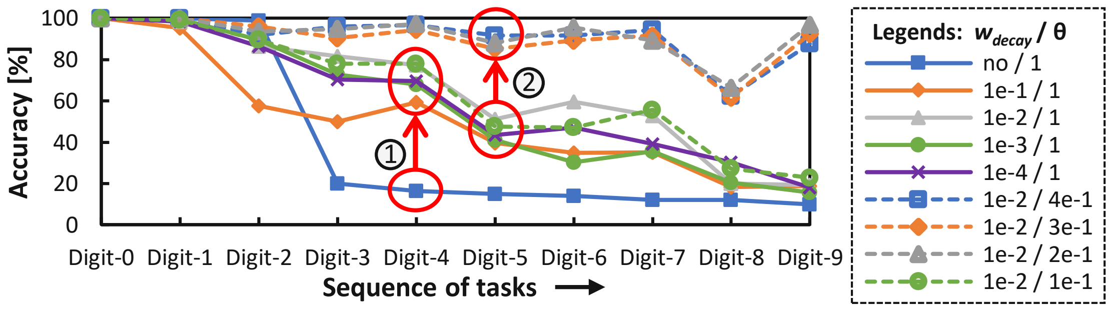


Figure 4.48: Impact of employing weight decay and adaptation potential $\theta$ on the accuracy of learning new tasks in a dynamic scenario.

**Adaptive Membrane Threshold Potential:** The neurons' threshold potential $v_{th}$ is defined by $v_{th} + \theta$, as discussed in Section 2.3.4. We observe that the adaptation potential $\theta$ has an important role to determine whether a neuron would generate spikes easily for later inputs. If $\theta$ is too high, the neurons will not spike easily for later inputs, since the threshold potential is already adjusted for recognizing earlier inputs. In the context of dynamic scenarios, the network will face difficulties when learning new tasks. If $\theta$ is too low, the neurons will spike easily for any inputs. In the context of dynamic scenarios, the network will quickly forget old information. Thus, the threshold potential should be balanced, so that some neurons are available for recognizing new features, while others retain the old yet significant information. Towards this, we define the adaptation potential $\theta$ to be proportional to its decay rate $\theta_{decay}$ and the presentation time of a sample $t_{sim}$, and can be stated as $\theta = c_\theta \cdot \theta_{decay} \cdot t_{sim}$, with $c_\theta$ denotes the adaptation constant. An appropriate $\theta$ can improve accuracy, as shown by the label-② in Figure 4.48.

**Reducing the Spurious Weight Updates:** Previous work [S+17] has observed that there are spurious updates in SNNs, which can degrade the accuracy. These are observed in two cases: (1) when the neurons spike unpredictably, due to the random weight initialization; and (2) when a neuron generates spikes for patterns that belong to different classes due to the overlapped features. We exploit this observation in a novel way to reduce the spurious updates that are induced by both the presynaptic and postsynaptic spikes. The idea is to employ a timestep period, and then monitor whether at least one postsynaptic spike happens. If so, then the weight potentiation will be conducted, and otherwise, the weight depression will be conducted (see Figure 4.49).

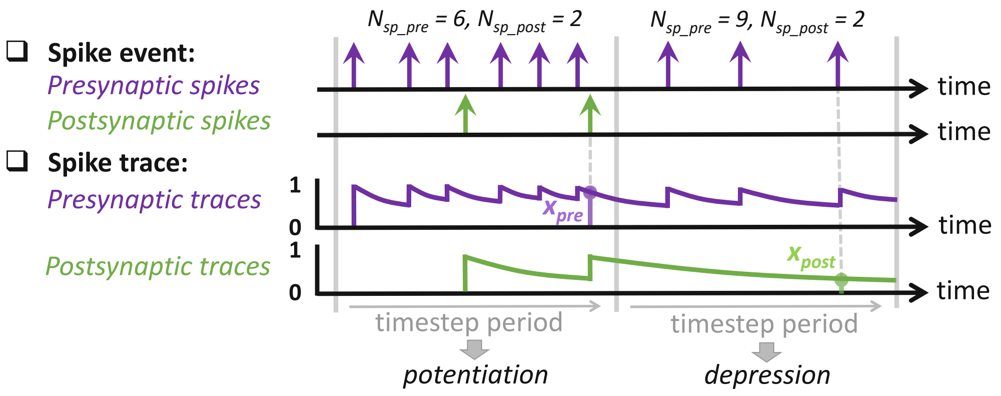


Figure 4.49: Overview of the proposed timestep-based weight updates during the STDP learning process.

### 4.5.5 Evaluation Methodology

Figure 4.50 shows the experimental setup for evaluating the SpikeDyn framework. We use Python-based SNN simulations [HSK$^+$18] that run on Embedded GPU (Nvidia Jetson Nano) and GPGPUs (Nvidia GTX 1080 Ti and RTX 2080 Ti) to perform diverse evaluations under different memory and compute capabilities, for showing the generality of our SpikeDyn framework. Note, GPU specifications are presented in Table 4.3.

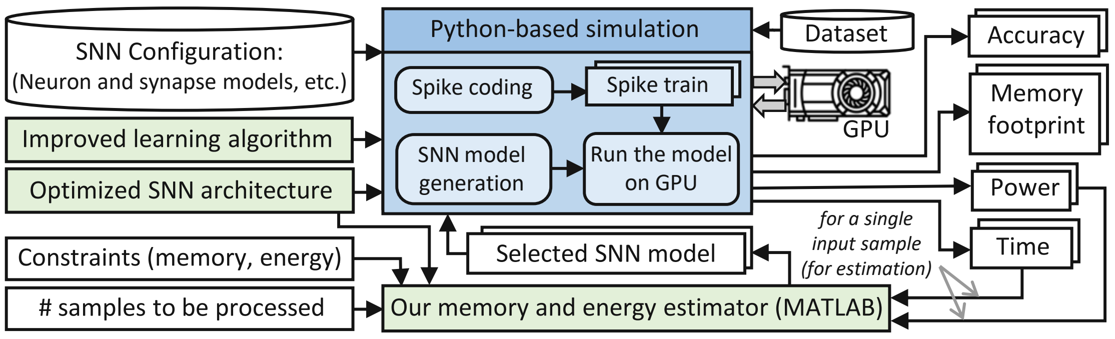

SNN Configuration:
(Neuron and synapse models, etc.)
Improved learning algorithm
Optimized SNN architecture
Constraints (memory, energy)
# samples to be processed
Python-based simulation
Spike coding
Spike train
SNN model generation
Run the model on GPU
Dataset
GPU
Accuracy
Memory footprint
Power
Time
Selected SNN model
for a single input sample (for estimation)
Our memory and energy estimator (MATLAB)

Figure 4.50: The experimental setup and tool flow.

Table 4.3: GPU Specifications.

| Category | Jetson Nano [Nvid] | GTX 1080 Ti [Nvib] | RTX 2080 Ti [Nvic] |
|---|---|---|---|
| Architecture | Maxwell | Pascal | Turing |
| CUDA cores | 128 | 3584 | 4352 |
| Memory | 4GB LPDDR4 | 11GB GDDR5X | 11GB GDDR6 |
| Interface width | 64-bit | 352-bit | 352-bit |
| Power | 10W | 250W | 250W |

To estimate the energy consumption of both the training and the inference phases, we adopt the approach of work in [HMD16]. That is, leveraging the information of the processing time, and the processing power that is reported through (1) *nvidia-smi* utility for GPGPUs, and (2) measurement using a power-meter for Embedded GPU. We use the MNIST as it is widely used for evaluating the continual and unsupervised learning in SNNs [PARR18, AR16, AR20], and employed the rate coding to convert each pixel of an image into a Poisson-distributed spike train. For comparison partners, we employ techniques in [DC15] as the baseline, and the adaptive synaptic plasticity (ASP) [PARR18] as the state-of-the-art since it is the only available recent work that has the complete set of configurations, parameters, and implementations details to have a reproducible design and results. The evaluation is performed for both dynamic and non-dynamic environments. *Dynamic environments* mean that the network is fed with consecutive task changes without re-feeding previous tasks, and each task has the same number of samples. It simulates an extreme condition where an SNN system receives training tasks from the environment in a consecutive manner, and each task has a defined number of samples. Meanwhile, *non-dynamic environments* mean that the network is fed with input samples whose tasks are distributed randomly. We consider networks with 200 and 400 excitatory neurons, which we refer to as N200 and N400, respectively.

### 4.5.6 Results and Discussion

#### 4.5.6.1 Maintaining Accuracy in Dynamic and Non-Dynamic Environments

**Dynamic Environments:** We evaluate the classification accuracy for two cases, i.e., (1) classifying the most recently learned task, which represents the capability of learning new information; and (2) classifying the previously learned tasks, which represents the capability of retaining old information.

*Case*-1: Figure 4.51(a.1) and Figure 4.51(b.1) show the accuracy when the network classifies the most recently learned task (i.e., learning new information) for N200 and N400, respectively. Here, the ASP achieves better accuracy than the baseline, since it employs an adaptive learning rate and weight decay, while the baseline does not consider the dynamic tasks in its learning. Meanwhile, our SpikeDyn improves learning capabilities more than the ASP, i.e., improving the accuracy by up to 38% (avg. 23%) than the ASP for N200, and by up to 29% (avg. 21%) for N400. The reason is that, the SpikeDyn employs: (1) a more careful mechanism to determine the rates of weight potentiation and depression for learning new features, (2) an appropriate neurons' threshold potential to adjust some neurons to be active in the learning process, (3) weight decay rate that effectively removes the old and insignificant information, and (4) reduction of the spurious weight updates.

*Case*-2: Figure 4.51(a.2) and Figure 4.51(b.2) show the accuracy when the network classifies the previously learned tasks (i.e., retaining old information) for N200 and N400, respectively. The baseline performs the worst as it does not decrease the weights, thereby suffering from mixed information in its synapses. Here, our SpikeDyn shows comparable accuracy to the ASP. The SpikeDyn improves the accuracy by up to 36% (avg. 4%) than the ASP for N200, and by up to 37% (avg. 8%) for N400. The reason is that, the

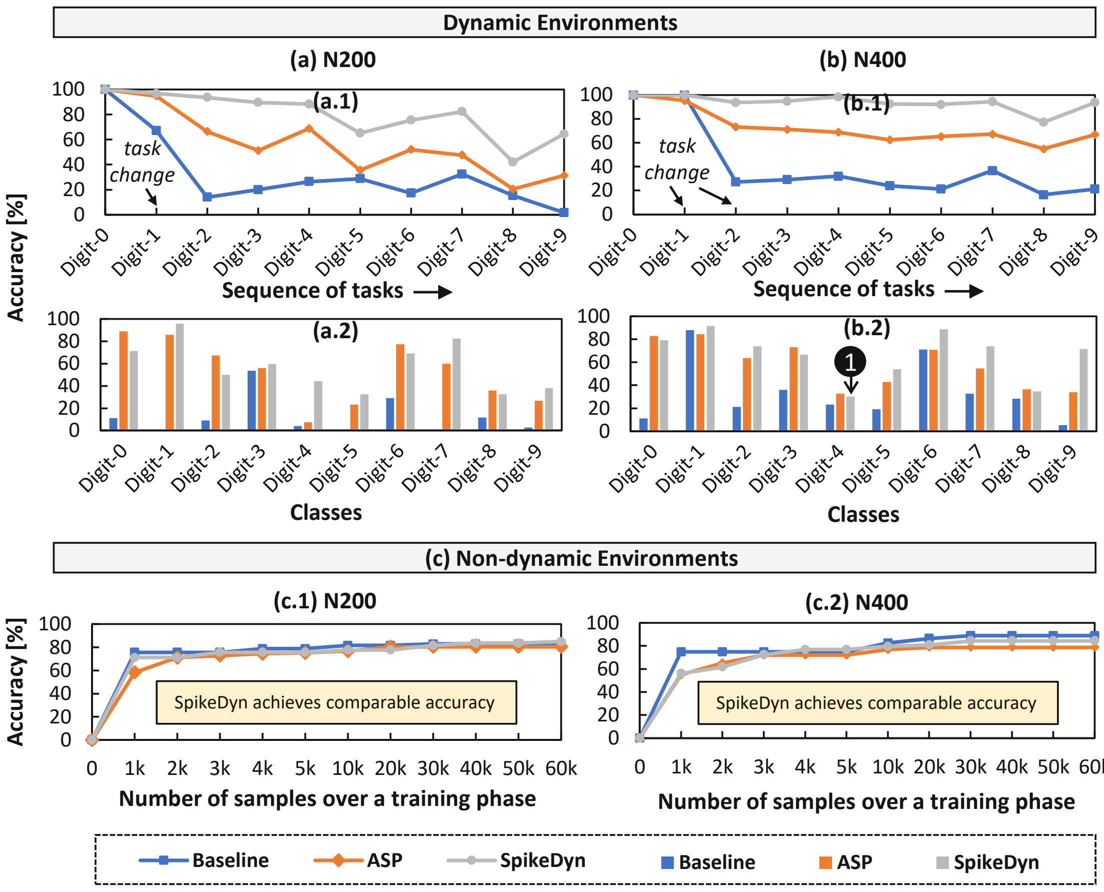


Figure 4.51: Accuracy in the *dynamic environments*: for most recently learned task in (a.1) N200, (b.1) N400; and for the previously learned tasks in (a.2) N200, (b.2) N400. Accuracy in the *non-dynamic environments*: over the presentation of training samples in (c.1) N200 and (c.2) N400.

SpikeDyn employs: (1) a neurons' threshold potential that tunes some neurons to be inactive in the learning process, thereby retaining the old yet significant information, and (2) weight decay rate that does not remove the old yet significant connections. Furthermore, we also observe that, some classes are relatively difficult to learn in dynamic scenarios, especially in the case of retaining old information. For instance, in N400 case, the accuracy for classifying digit-4 is low, as indicated by label-❶ in Figure 4.51(b.2). It is because a considerable number of misclassification happens when the digit-4 is recognized as another digit (i.e., digit-9), as indicated by label-❷ in Figure 4.52(b). The reason is that, the learned features from digit-4 are gradually changed to represent the features of digit-9 over a training period, due to their overlapped features and the sequence of learning tasks. Therefore, some neurons that recognize digit-4 at the early of the training period, are changed to recognize digit-9 at the end of the training period.

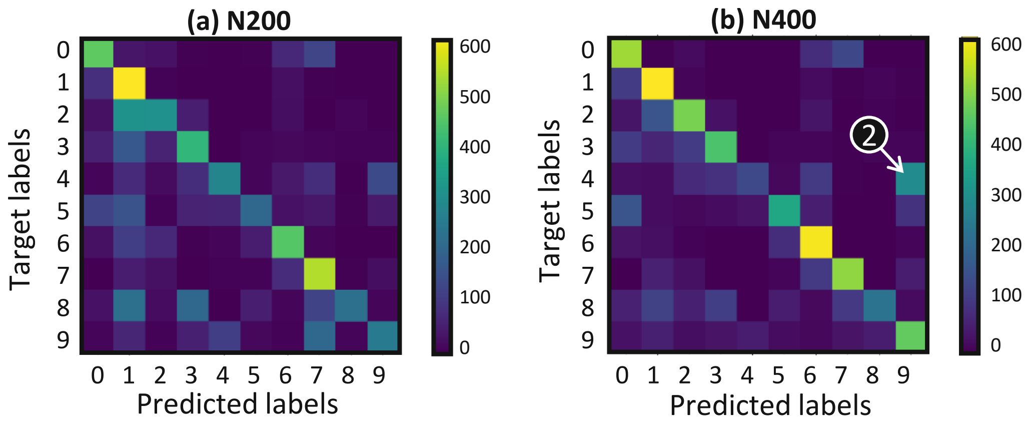


Figure 4.52: Confusion matrices of the SpikeDyn for classifying the previously learned tasks, which show the relation between the target labels and the predicted labels in (a) N200 and (b) N400.

**Non-dynamic Environments:** Figure 4.51(c.1) and Figure 4.51(c.2) show the accuracy over the presentation of training samples for N200 and N400, respectively. The results show that our SpikeDyn achieves comparable accuracy to other techniques. The reason is that, our SpikeDyn employs effective learning rates to potentiate and decrease the weights, while reducing the spurious updates. In this manner, each weight update is adjusted appropriately, and hence the accuracy is maintained. Such observations are important, as SNN systems may have a different number of training samples available from the environment. Therefore, the users can devise a strategy to define the minimum training samples for achieving the targeted accuracy.

#### 4.5.6.2 Reduction of Energy Consumption

Figure 4.53 shows that our SpikeDyn reduces energy consumption as compared to other techniques, across different sizes of networks and different GPUs, for both dynamic and non-dynamic environments. For N200, our SpikeDyn reduces the energy consumption from the ASP by up to 59% (avg. 57%) for training, and by up to 54% (avg. 51%) for inference. For N400, our SpikeDyn reduces the energy consumption from the ASP by up to 66% (avg. 51%) for training, and by up to 54% (avg. 37%) for inference. The energy savings in training come from the elimination of the inhibitory neurons, the reduction of spurious updates as well as exponential calculations. Meanwhile, the energy savings in inference mainly come from the elimination of inhibitory neurons. Furthermore, we also observe the processing time of the SpikeDyn for the training and inference phases (see Table 4.4). The results show that running an SNN model on the Embedded GPU (Jetson Nano) requires a longer time than the GPGPUs, as the Embedded GPU has less number of cores, memory size, and bandwidth. Therefore, the users should devise a strategy for defining the number of samples in the training and inference phases, to comply with the use-cases' requirements, especially for the embedded applications.

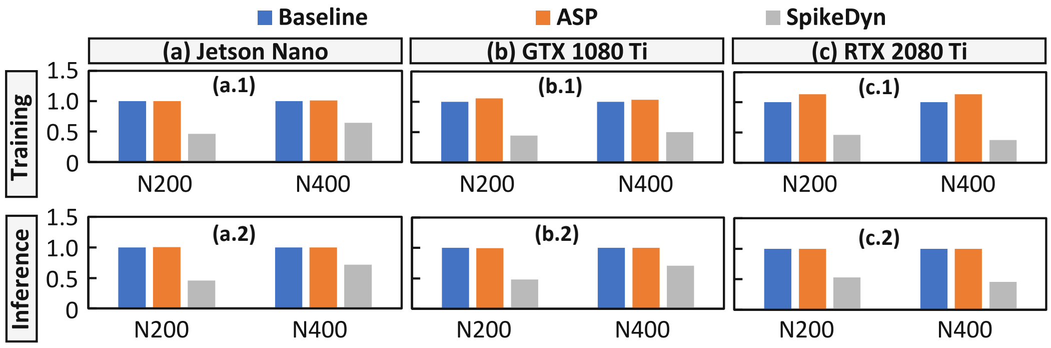


Figure 4.53: The energy consumption (normalized to the baseline) for the training and inference phases, and across different sizes of networks and different GPUs.

Table 4.4: Processing tome of the SpikeDyn on *full* MNIST dataset.

| **Process** | **Jetson Nano** | | **GTX 1080 Ti** | | **RTX 2080 Ti** | |
|---|---|---|---|---|---|---|
| | **N200** | **N400** | **N200** | **N400** | **N200** | **N400** |
| Training (hours) | 35.0 | 36.3 | 5.0 | 5.3 | 3.9 | 4.1 |
| Inference (hours) | 4.7 | 4.8 | 0.7 | 0.7 | 0.6 | 0.6 |
| Inference of an image (seconds) | 1.71 | 1.74 | 0.25 | 0.25 | 0.2 | 0.2 |

### 4.5.7 Summary of SpikeDyn Framework

We propose a novel SpikeDyn framework that supports a lightweight unsupervised continual learning for SNNs while reducing their energy consumption, by optimizing the SNN operations and improving the learning algorithm. The experimental results show that our SpikeDyn incurs less energy consumption and improves accuracy, as compared to the state-of-the-art in both dynamic and non-dynamic environment scenarios. Therefore, our SpikeDyn may enable energy-efficient embedded SNNs with one-time deployment.

## 4.6 lpSpikeCon: Enabling Low-Precision SNNs in Unsupervised Continual Learning Settings

This section aims at addressing **Problem-5** with the solution for efficiently implementing low-precision SNNs with unsupervised continual learning capabilities under tight memory constraints.

### 4.6.1 Motivational Study

To understand the accuracy profile of SNNs with low-precision weights under dynamic environment settings, we perform an experimental case study. Here, we perform experiments that provide dynamic environment scenarios to the network by feeding consecutive tasks/classes using training samples, train the network accordingly, then evaluate the

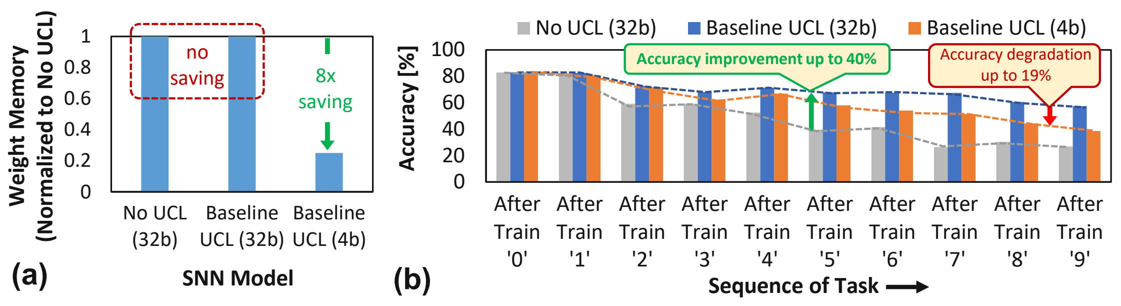


Figure 4.54: (a) Weight memory of a 400-neuron network with different learning conditions: No Unsupervised Continual Learning (No UCL) with 32-bit weights; Baseline UCL with 32-bit weights, adapted from [PS21b]; and Baseline UCL with 4-bit weights, using quantization. (c) Accuracy of a 400-neuron network with different learning conditions. In this work, we consider the UCL algorithm from the work of [PS21b].

trained network using the test samples for tasks/classes that have been fed so far. Following are the steps of experiments using the MNIST dataset.

- First, we feed a stream of training samples for digit-0, and train the network accordingly. Then, we evaluate the trained network using the test samples for digit-0.
- Second, we repeat the above steps but for training digit-1, then evaluate the trained network using the test samples for digit-0 and digit-1.
- The above steps are repeated but for training another digit, and testing the digits that have been learned so far, until all 10 digits in the MNIST dataset are used for training and testing.

Our experiments consider the fully-connected network architecture shown in Figure 4.46(a) and different learning conditions: (1) No Unsupervised Continual Learning (No UCL); (2) Baseline UCL with 32-bit weights, adapted from [PS21b]; and (3) Baseline UCL with 4-bit weights. Further details of the experimental setup are presented in Section 4.6.5. The experimental results are shown in Figure 4.54(b)-(c), from which we make the following observations.

- The unsupervised continual learning improves the accuracy under dynamic scenarios, due to its carefully crafted weight potentiation/depression strategy to learn new features, while retaining old yet important ones.
- Reduction of weight precision can significantly save the SNN weight memory, e.g., reducing precision from 32-bit to 4-bit weights enables 8x weight memory saving. However, it may degrade the quality of unsupervised continual learning due to knowledge/information loss.
- A network with 4-bit weights and continual learning may achieve higher accuracy than a network with 32-bit weights but no continual learning, thereby showing the potential of memory reduction for a network under dynamic scenarios.

### 4.6.2 Scientific Research Challenges

These observations highlight the following key challenges to devise an efficient solution for the targeted problem.

- *Quantization should be performed judiciously to remove non-significant information in each weight*, hence retaining most of the important information and maintaining the learning quality (i.e., high accuracy).
- *The solution should employ simple yet effective enhancements* to compensate for the information loss due to weight quantization, thus enabling energy-efficient learning.

***Required:*** *An optimization technique that effectively performs unsupervised continual learning for SNNs with quantized weights under dynamic environments, thus enabling their implementation on tightly-constrained embedded systems (e.g., autonomous agents).*

### 4.6.3 Novel Contributions

To address the above challenges, we propose *lpSpikeCon, a novel methodology that enables low-precision Spiking neural network processing for efficient unsupervised Continual learning.* To the best of our knowledge, this work is the first effort that aims at reducing the weight precision of SNNs, while maintaining the quality of unsupervised continual learning under dynamic scenarios. Following are the key steps of our lpSpikeCon methodology (see an overview in Figure 4.55).

- **Analyzing the characteristics of SNN accuracy profiles** for each given task under different quantization levels and unsupervised continual learning settings.
- **Identifying the SNN parameters that have a significant impact on the accuracy**. It leverages the accuracy analysis to determine SNN parameters and their adjustment rules to get better neuronal dynamics for unsupervised continual learning, and hence the accuracy.
- **Devising an algorithm for determining SNN parameter values** that effectively improve the learning quality. It leverages the parameter adjustment rules for guiding the algorithm to refine the parameter values for achieving high accuracy in dynamic scenarios/environments.

### 4.6.4 lpSpikeCon Methodology

lpSpikeCon methodology employs the following steps [PS22a] (overview in Figure 4.56).

1. **Analyzing the SNN accuracy profiles (Section 4.6.4.1)** through the following.
   - Quantizing the weights of a given network.
   - Training the network with quantized weights using unsupervised continual learning under dynamic scenarios.
   - Observing the accuracy of each given task/class under different quantization levels.

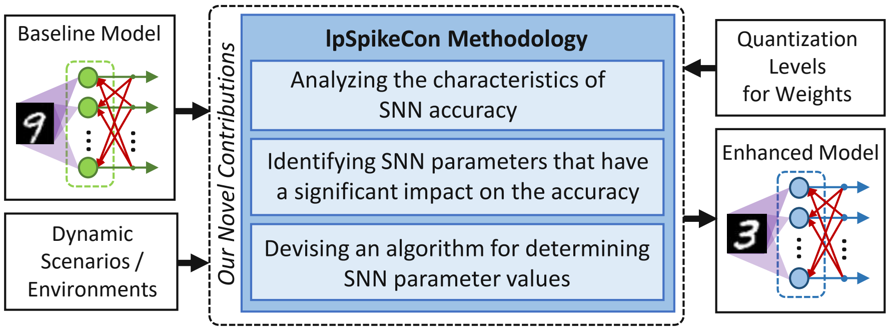

Baseline Model
Dynamic Scenarios / Environments
Our Novel Contributions
lpSpikeCon Methodology
Analyzing the characteristics of SNN accuracy
Identifying SNN parameters that have a significant impact on the accuracy
Devising an algorithm for determining SNN parameter values
Quantization Levels for Weights
Enhanced Model

Figure 4.55: The lpSpikeCon methodology. Its novel components are highlighted in blue.

2. **Identifying the SNN parameters and their adjustment rules for unsupervised continual learning (Section 4.6.4.2)** through the following.
   - Leveraging the accuracy analysis to select SNN parameters that have a significant impact on accuracy.
   - Devising the adjustment rules for the selected parameters (e.g., threshold potential $v_{th}$) to derive better neural dynamics for unsupervised continual learning.
3. **Devising an algorithm that refines SNN parameter values for the learning process (Section 4.6.4.3)**. It is done by leveraging the adjustment rules to gradually increase/decrease parameter values that improve the learning quality.

Note, the lpSpikeCon-enhanced SNN systems need to be scheduled for updating their offline-trained knowledge regularly at run time. The update is performed through online training using data gathered from the operational environments. After completing the training mode, the systems are back to the inference mode. Here, the online training can be scheduled based on user-defined mechanisms (e.g., the online training is performed with training samples each time the system has performed inference for a certain number of input samples). In this manner, the SNN systems are expected to adapt better to diverse operational environments than the offline-trained ones.

#### 4.6.4.1 Analyzing the SNN Accuracy Profiles

To devise a lightweight solution that enables efficient unsupervised continual learning for a quantized SNN, *it is important to understand the characteristics of SNN accuracy for each given task/class under different quantization levels and dynamic scenarios*. To do this, we perform the following steps.

- We quantize the weights of a given SNN model using the truncation approach, as described in Section 2.5.1.
- Then, we train the quantized model using unsupervised continual learning under dynamic scenarios. To do this, we feed consecutive tasks/classes using the training samples, train the model accordingly, then evaluate the trained model using the test samples for tasks/classes that have been fed so far. Pseudo-code of this step is also presented in Algorithm 11.

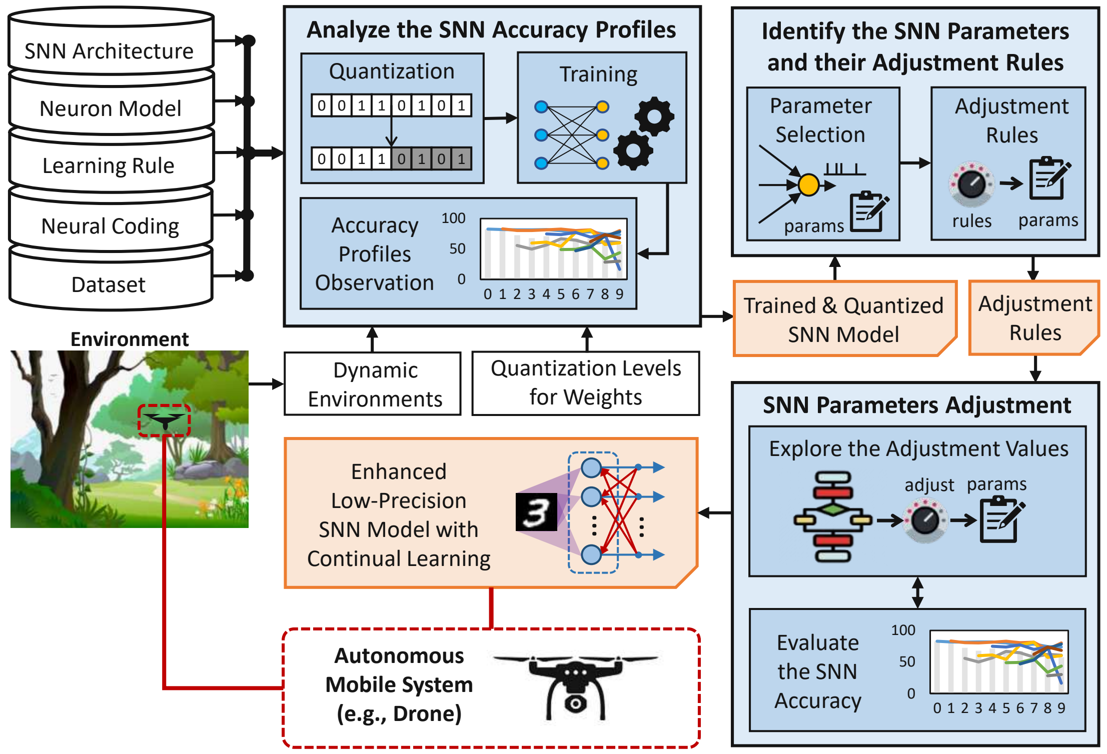

SNN Architecture
Neuron Model
Learning Rule
Neural Coding
Dataset
Environment
Analyze the SNN Accuracy Profiles
Quantization
0 0 1 1 0 1 0 1
0 0 1 1 0 1 0 1
Training
Accuracy Profiles Observation
100
50
0
0 1 2 3 4 5 6 7 8 9
Identify the SNN Parameters and their Adjustment Rules
Parameter Selection
params
Adjustment Rules
rules
params
Trained & Quantized SNN Model
Adjustment Rules
Dynamic Environments
Quantization Levels for Weights
SNN Parameters Adjustment
Explore the Adjustment Values
adjust
params
Enhanced Low-Precision SNN Model with Continual Learning
Evaluate the SNN Accuracy
100
50
0
0 1 2 3 4 5 6 7 8 9
Autonomous Mobile System (e.g., Drone)

Figure 4.56: An overview of the lpSpikeCon methodology. The novel contributions are highlighted in blue.

- Afterward, we observe the profiles of inference accuracy for each given task/class under different levels of weight quantization, to understand the sources of accuracy degradation and get insights on how to address it.

The experimental results are presented in Figure 4.57, from which we make the following key observations.

- In general, reduced weight precision degrades the accuracy of unsupervised continual learning due to information loss. The accuracy degradation is noticeable especially for tasks that are learned at the later training sequence, as shown by labels ①, ②, and ③ in Figure 4.57.
- Lower weight precision also leads to lower accuracy for more tasks. For instance, a model with 4-bit weights suffers from very low accuracy ($\leq 20\%$ accuracy) on four tasks, i.e., digit-6, digit-7, digit-8, and digit-9 (see label ③ in Figure 4.57), while a model with 8-bit weights suffers from very low accuracy only on task digit-9 (see label ① in Figure 4.57).

These observations lead to several insights for devising an efficient solution for the targeted problem, as discussed in the following.

**Algorithm 11** Training and testing under dynamic scenarios

```
INPUT: (1) SNN model ($model_{in}$);
    (2) Dataset ($D$): dataset for class-$i$ ($D[i]$), training set for class-$i$ ($D[i].train$), testing set for
    class-$i$ ($D[i].test$);
OUTPUT: (1) Trained SNN model ($model_{out}$);
    (2) Accuracy ($acc$)
BEGIN
    Initialization:
 1: $model_{out} = model_{in}$;
    Process:
 2: $task = class(D)$
 3: for $i \in task$ do
 4:     $model_{out} \leftarrow \text{train}(model_{out}, D[i].train)$;
 5:     for $k = 0$ to $i$ do
 6:         $acc[i,k] = \text{test}(model_{out}, D[k].test)$;
 7: return $model_{out}$;
END
```

- Reduced weight precision has a less memory capacity for storing unique information from new tasks, thereby making the quantized SNN model difficult to recognize novel features from new tasks.
- Different SNN models with different levels of weight precision have different accuracy profiles. Hence, such models require specialized settings for achieving high accuracy.

The above discussion indicates that, the potential solution is *employing parameter adjustments that provide more available memories for storing novel information from new tasks and consider specialized settings for different levels of weight precision.* Hence, costly additional components and/or operations can be avoided, leading to efficient learning.

#### 4.6.4.2 Identifying SNN Parameters and Their Adjustment Rules

The analysis in Section 4.6.4.1 suggests that the efficient solution is by employing parameter adjustments for the given quantized SNN model. Therefore, *it is important to identify SNN parameters that have a significant impact on accuracy, and their effective adjustment rules for improving learning quality of the quantized model.*

**SNN Parameters:** To identify SNN parameters that should be adjusted for better learning quality, we leverage the analysis from Section 4.6.4.1. The analysis shows that the reduced weight precision makes the model difficult to learn tasks that appear at the later training sequence. This means that the weight bits are strongly associated with the previously learned tasks, and not flexible enough to change their context to another task. To address this, *we adjust the neuronal dynamics so that the synaptic plasticity becomes more flexible for learning new tasks, especially when low weight precision is employed.* To do this, we adjust two parameters to change the flexibility of neuronal dynamics

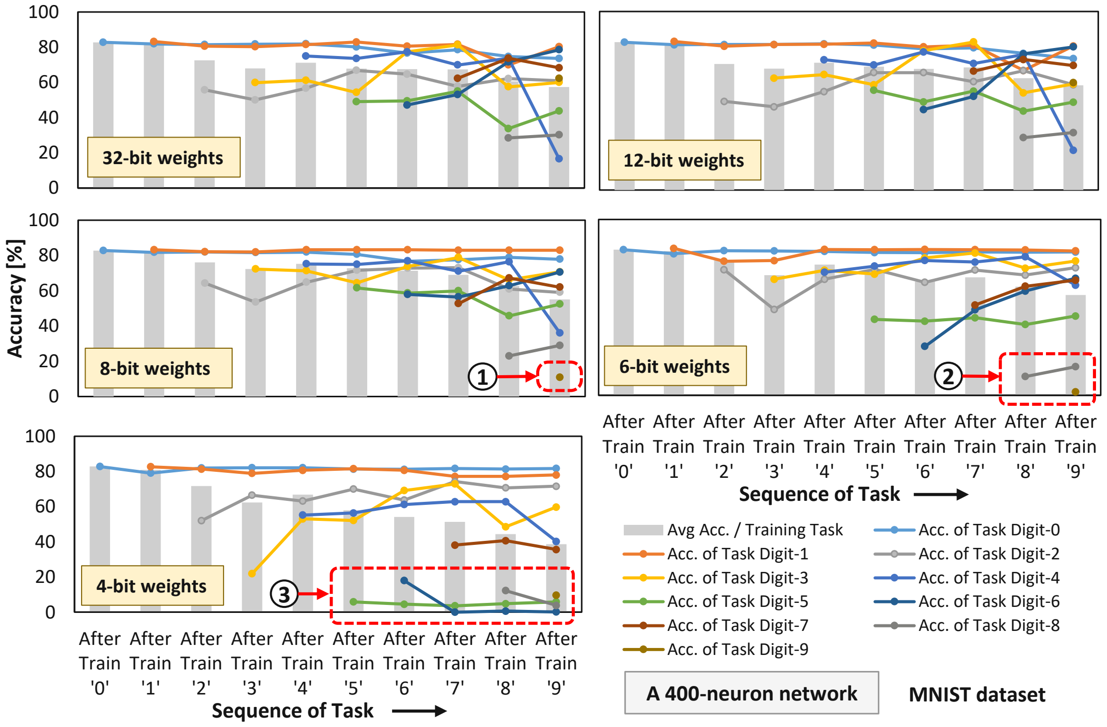


Figure 4.57: The accuracy profiles of a 400-neuron network for the MNIST dataset under different levels of weight precision (i.e., 32, 12, 8, 6, and 4 bits) and dynamic scenarios. The colored line represents the accuracy for each task/class throughout the consecutive training phases from digit-0 to digit-9. The grey bar represents the average accuracy after each training phase of a task/class.

for learning activity: *adaptive membrane threshold potential* ($v_{th}$) and *weight decay rate* ($w_{decay}$). This selection is based on the following reasons.

- $v_{th}$ determines how far $v_{mem}$ should be increased to generate a spike, which then triggers weight potentiation for learning. The distance between $v_{th}$ and $v_{mem}$ is inversely proportional to the frequency for learning activity.
- $w_{decay}$ determines how fast each synaptic weight is depressed for removing the learned information, and providing available memory for storing novel information from new tasks (classes).

**Adjustment Rules:** To properly adjust the selected SNN parameters (i.e., $v_{th}$ and $w_{decay}$) for achieving better learning quality of a quantized SNN model, we propose the following *parameter adjustment rules.*

- In the non-quantized model, $v_{th}$ is set with a value that prevents catastrophic forgetting [CL18, PKP+19, MC89, PS21b]. Therefore, *to make $v_{th}$ more suitable for the quantized model, its value should be less or equal ($\leq$) than $v_{th}$ of the non-quantized model.* In this manner, each neuron is expected to reach $v_{th}$ faster and to generate spikes more frequently, which triggers learning activity for any incoming tasks.

- In the non-quantized model, $w_{decay}$ is set with a value that provides available memory for storing learned information from new tasks. Hence, *to make $w_{decay}$ more suitable for the quantized model, its value should be greater or equal ($\geq$) than $w_{decay}$ of the non-quantized model.* In this manner, each weight is expected to decay faster, hence providing available memory for storing learned features from any incoming tasks.

To justify these rules, we perform an experimental case study to see the impact of our adjustment rules, i.e., "decreased $v_{th}$" and "increased $w_{decay}$". Experimental results are provided in Figure 4.58. These results show that our adjustment rules improve the quality of unsupervised continual learning for both "decreased $v_{th}$" and "increased $w_{decay}$" cases, since each case has less number of tasks with very low accuracy, i.e., $\leq 20\%$ accuracy (see labels ④ and ⑤ in Figure 4.58), as compared to the model with 4-bit weights and baseline continual learning (see label ③ in Figure 4.57).

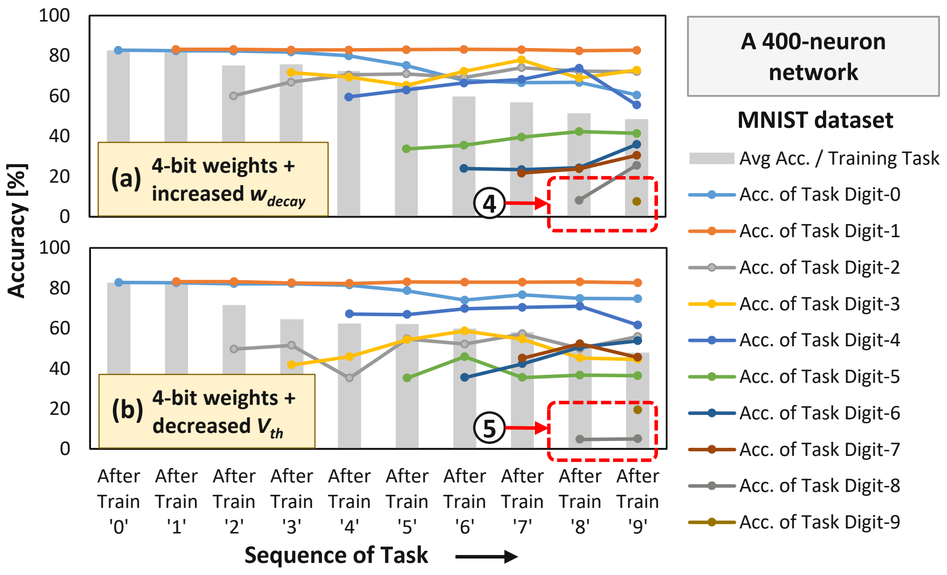


Figure 4.58: The accuracy profiles of a 400-neuron network with 4-bit weights on the MNIST dataset under dynamic scenarios and different parameter adjustments. (a) "Increased $w_{decay}$" by using baseline $w_{decay} + 0.09$. (b) "Decreased $v_{th}$" by using $v_{th}+$ (baseline $\theta - 0.2$). These adjustment values are manually selected to clearly see their impacts on accuracy.

### 4.6.4.3 An Algorithm for Refining SNN Parameter Values

To properly adjust the values of the selected SNN parameters, a systematic mechanism is required. Toward this, *we propose an algorithm that leverages our adjustment rules to refine the selected SNN parameter values ($v_{th}$ and $w_{decay}$) for achieving better learning quality, especially under dynamic scenarios.* This algorithm is developed based on the following ideas, and the detailed steps are provided in Algorithm 12.

- The range of values for each selected SNN parameter ($v_{th}$ or $w_{decay}$) should be defined for guiding the design space exploration. The lower-bound value for $v_{th}$ is $v_{th}^l$, while the upper-bound value for $w_{decay}$ is $w_{decay}^u$.
- The accuracy for each task ($acc_{task}$) should not be less or equal ($\leq$) than the defined value. Therefore, if we consider $acc_{low}$ as the borderline of low accuracy, then the acceptable accuracy for each task is defined as follows.

$$acc_{task} > acc_{low} \tag{4.16}$$

- The average accuracy of the lpSpikeCon-enhanced SNN model ($acc_{avg}^*$) should be within an acceptable accuracy loss. Hence, if we consider $acc_{avg}$ as the average accuracy of the non-quantized SNN model with baseline unsupervised continual learning, and $acc_{loss}$ as the acceptable accuracy loss, then the $acc_{avg}^*$ is defined as follows.

$$acc_{avg}^* \geq (acc_{avg} - acc_{loss}) \tag{4.17}$$

**Algorithm 12** Adjustment steps for SNN parameter values

```
INPUT: (1) SNN: baseline non-quantized model ($model_{in}$), weight decay rate ($w_{decay}$), upper-
    bound of $w_{decay}$ ($w_{decay}^u$);
    (2) Exploration variables: investigated/evaluated model ($model_{eval}$), increasing step for
    $w_{decay}$ ($step\_w$), decreasing step for $v_{th}$ ($step\_v_{th}$);
    (3) Functions: accuracy for each task ($acc_{task}$), average accuracy of the given model ($acc$),
    acceptable accuracy loss ($acc_{loss}$);
OUTPUT: Trained SNN model ($model_{out}$);
BEGIN
    Initialization:
 1: $model_{out} = model_{in}$;
    Process:
 2: for ($p = w_{decay}$;  $p \leq w_{decay}^u$;  $p = p + step\_w$) do
 3:     for ($q = v_{th}$;  $q \geq v_{th}^l$;  $q = q - step\_v_{th}$) do
 4:         $model_{eval} = model_{in}$;
 5:         update the values of selected SNN parameters;
 6:         perform training and test on $model_{eval}$ using Algorithm 11
 7:         if ($acc_{task}(model_{eval}) > acc_{low}$)  and  ($acc(model_{eval}) \geq (acc(model_{in}) - acc_{loss})$)
            then
 8:             if ($acc(model_{eval}) \geq acc(model_{out})$) then
 9:                 $model_{out} = model_{eval}$
10: return  $model_{out}$;
END
```

### 4.6.5 Evaluation Methodology

Figure 4.59 shows the experimental setup for evaluating the lpSpikeCon methodology. We employ a Python-based framework [HSK+18] that runs on a multi-GPU machine (i.e., Nvidia RTX 2080 Ti) and an Embedded-GPU machine (i.e., Nvidia Jetson Nano) to perform evaluations on different platforms with different memory and compute capabilities. We employ a single-layer fully-connected network shown in Figure 4.46(a) with different network sizes (i.e., 200 and 400 excitatory neurons), since it has shown the capabilities for performing unsupervised continual learning under resource- and power/energy-constrained embedded platforms [PS21b].

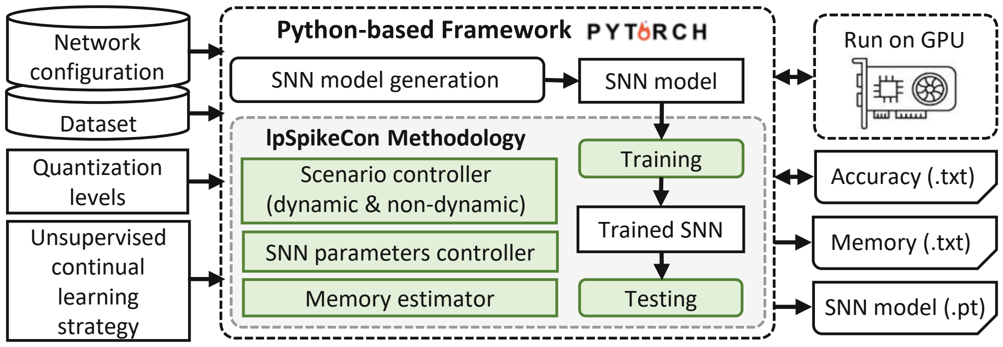


Figure 4.59: The experimental setup for evaluating our lpSpikeCon methodology.

We consider the MNIST dataset as workload, since it has been widely used for evaluating unsupervised continual learning in the SNN community [AR16, PARR18, AR20, PS21b]. For the baseline unsupervised continual learning, we consider the learning strategy from SpikeDyn [PS21b]. The evaluation is performed under both non-dynamic and dynamic environment scenarios. *Non-dynamic environment scenario* is provided by feeding the network with training samples whose tasks/classes are randomly distributed. It aims at simulating conventional offline training where all training samples are already available. Meanwhile, *dynamic environment scenario* is provided by feeding the network with consecutive tasks/classes, where each task/class has the same number of samples, and without re-feeding previous tasks/classes. It aims at simulating an extreme condition where the deployed SNN system receives training tasks in a consecutive manner from the environment at run time for updating its learned information/knowledge.

## 4.6.6 Results and Discussion

### 4.6.6.1 Maintaining Accuracy under Quantized Weights

**Dynamic Environment Scenario:** We evaluate accuracy considering different network sizes (i.e., 200 and 400 excitatory neurons) and different weight precision levels (i.e., 32, 16, 14, 12, 8, 6, and 4 bits). The experimental results are provided in Figure 4.60 and Figure 4.61 for a 200-neuron network and 400-neuron network, respectively. These results show that lower weight precision leads to lower accuracy for more recognition tasks due to information loss. For instance, in a 200-neuron network, 6-bit weights lead to very low accuracy (i.e., $\leq 20\%$) on one task (see label ❶), while 4-bit weights lead to very low accuracy on four tasks (see label ❷). Such patterns are also observed in a 400-neuron network, as shown by labels ❺ and ❻.

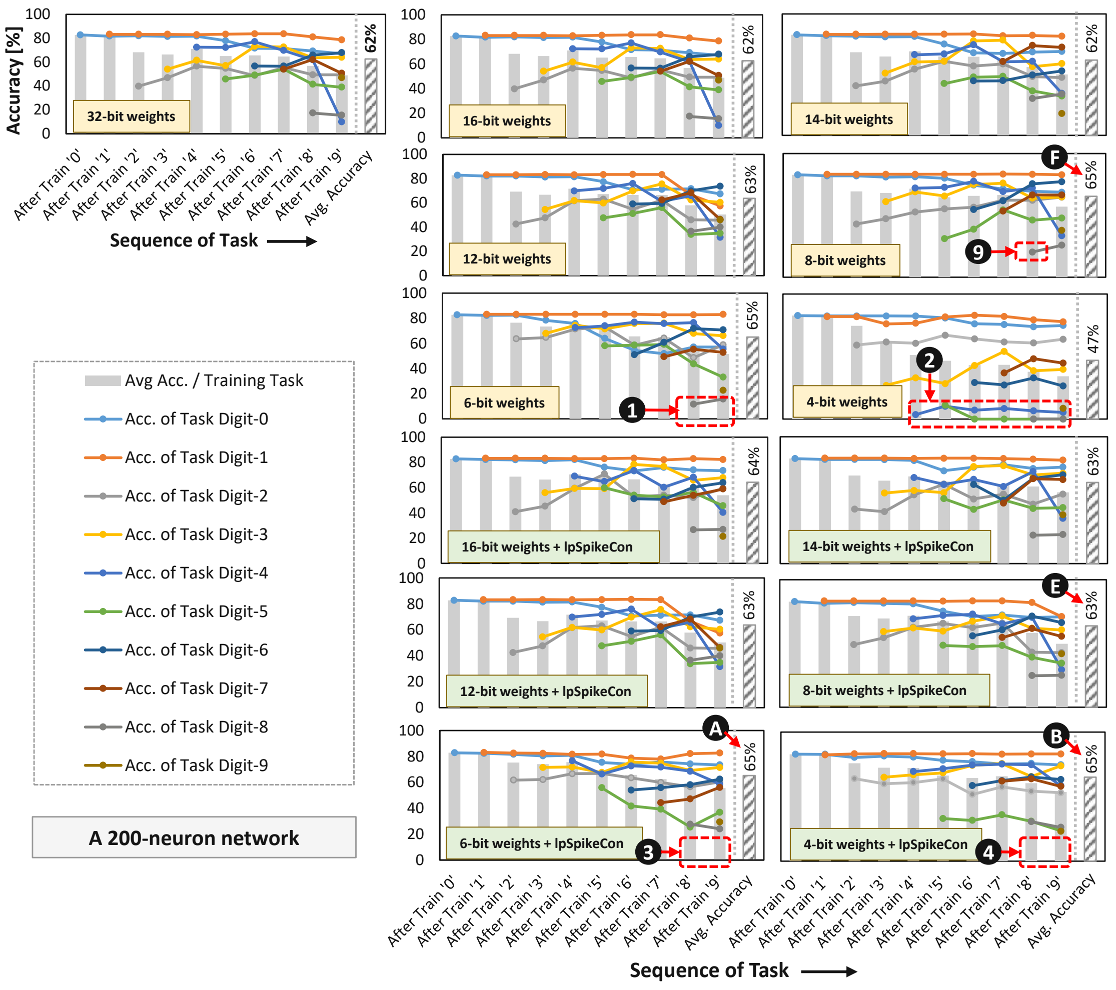


Figure 4.60: The accuracy profiles of a 200-neuron network under different levels of weight precision and dynamic scenarios. The colored line represents the inference accuracy for each task/class throughout the consecutive training phases. The grey-colored bar represents the average accuracy after each training phase of a task/class. The pattern-coded bar represents the average accuracy for all evaluated tasks/classes.

The experimental results also show that our lpSpikeCon methodology can improve the accuracy of the quantized SNNs, which can be observed in two aspects. First, the lpSpikeCon-enhanced SNNs do not suffer from very low accuracy (i.e., $\leq$ 20%) for recognizing any tasks/classes; see labels ❸ and ❹ for a 200-neuron network, and labels ❼ and ❽ for a 400-neuron network. Second, the lpSpikeCon-enhanced SNNs also achieve no accuracy loss on average when compared to the non-quantized SNNs with baseline unsupervised continual learning, across different network sizes. For instance, in a 200-neuron network, average accuracy for the 6-bit and 4-bit weights with lpSpikeCon is 65%, which is slightly higher than the 32-bit weights with baseline learning (i.e., 62%); see labels Ⓐ and Ⓑ. A similar pattern is also observed in a 400-neuron network, as

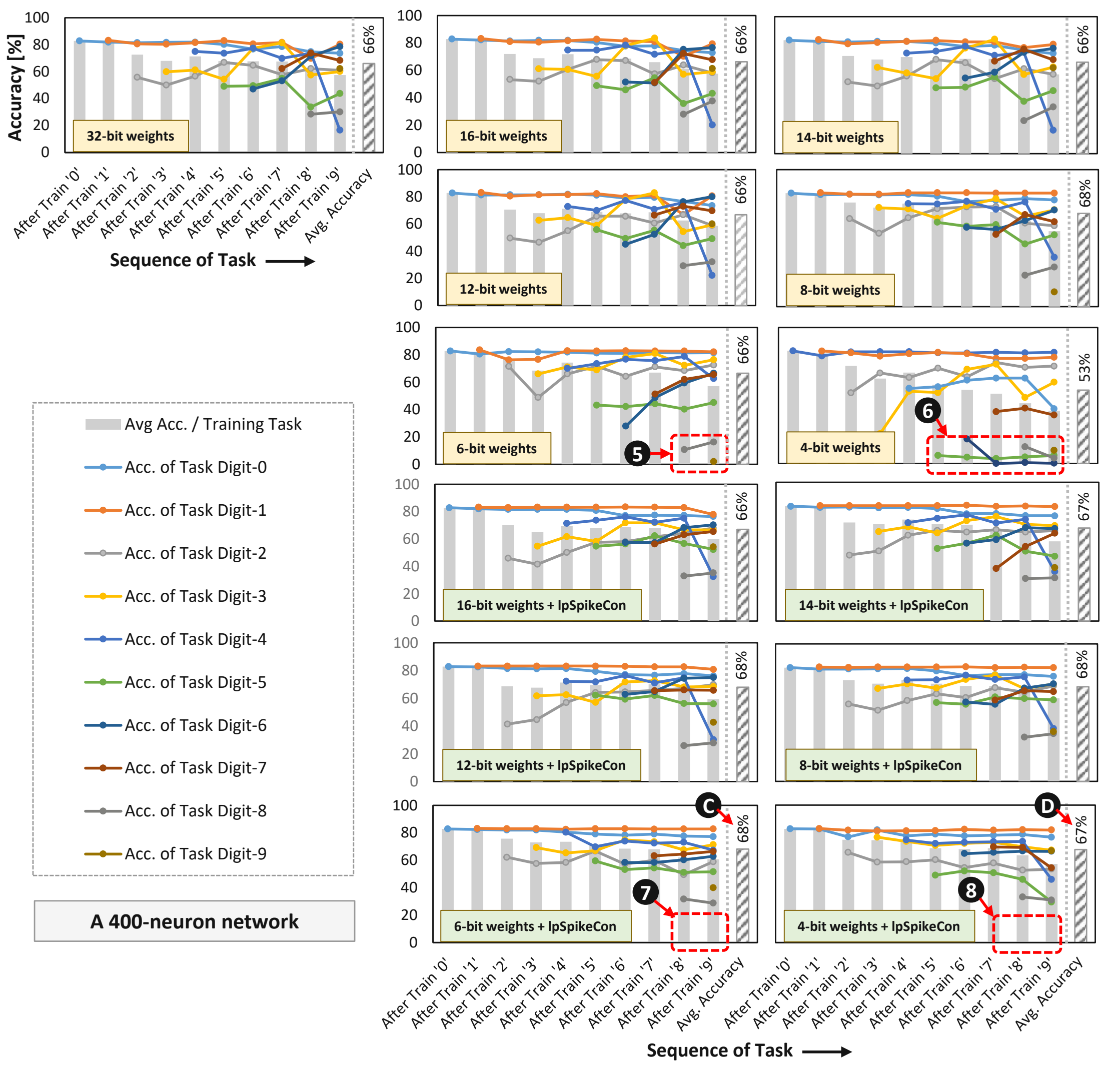


Figure 4.61: The accuracy profiles of a 400-neuron network, under different levels of weight precision and dynamic scenarios. The colored line represents the inference accuracy for each task/class throughout the consecutive training phases. The grey-colored bar represents the average accuracy after each training phase of a task/class. The pattern-coded bar represents the average accuracy for all evaluated tasks/classes.

the average accuracy for the 6-bit and 4-bit weights with lpSpikeCon are 68% and 67% respectively, which are slightly higher than the 32-bit weights with baseline learning (i.e., 66%); see labels C and D. These accuracy improvements are due to proper adjustments on the selected SNN parameters (i.e., $v_{th}$ and $w_{decay}$). These adjustments trigger the neuronal dynamics in the SNN system to quickly provide available memory for learning new tasks through the "increased $w_{decay}$" approach, and trigger learning activity for any incoming tasks (including the new tasks) through the "decreased $v_{th}$" approach. From the results, we also observe that in a certain case, the lpSpikeCon-enhanced model may have lower average accuracy than the non-enhanced model under the same weight precision. For instance, in the 200-neuron network with 8-bit weights, our lpSpikeCon-enhanced model achieves 63% accuracy (see label E), while the non-enhanced one achieves 65% (see label F). The reason is that, the lpSpikeCon considers the borderline of low accuracy ($acc_{low}$) as a constraint to determine the parameter adjustments and the output model. Therefore, since the non-enhanced model has lower accuracy than the defined $acc_{low}$ for one task, this model is not considered as the solution (see label 9).

**Non-Dynamic Environment Scenario:** We evaluate accuracy considering different network sizes (i.e., 200 and 400 excitatory neurons) and different weight precision levels (i.e., 32, 16, 14, 12, 8, 6, and 4 bits), and the experimental results are provided in Figure 4.62. These results show that, our lpSpikeCon-enhanced SNNs can achieve comparable accuracy to the non-enhanced SNN counterparts (i.e., SNNs that employ baseline unsupervised continual learning under the same weight precision). For instance, in the 400-neuron network with 6-bit weights, our lpSpikeCon-enhanced SNN achieves 78% accuracy and the non-enhanced one achieves 75% accuracy; while in the case of the 4-bit weight, the lpSpikeCon-enhanced SNN achieves 71% accuracy and the non-enhanced one achieves 77% accuracy, as highlighted by label G. The reason is that, our lpSpikeCon methodology performs exploration for parameter adjustments within a range of values that is close to the baseline settings. In this manner, proper adjustment values can be found fast, and these values are expected to preserve good characteristics from the baseline settings of unsupervised continual learning, such as achieving high accuracy under non-dynamic environment scenario.

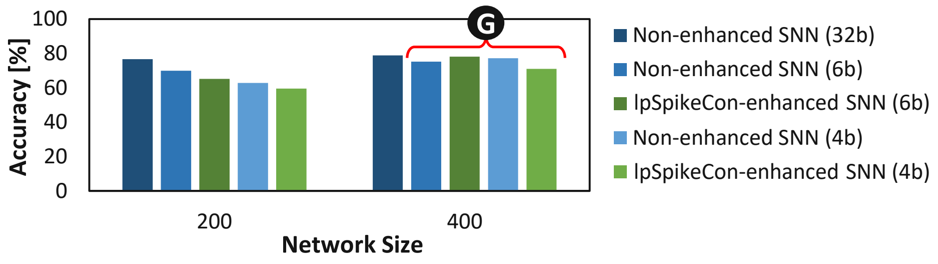


Figure 4.62: The accuracy of (a) a 200-neuron network, and (b) a 400-neuron network, under different levels of weight precision and non-dynamic scenarios. Here, the non-enhanced SNN refers to the model that employs baseline unsupervised continual learning.

#### 4.6.6.2 Weight Memory Savings for Efficient SNN Systems

Figure 4.63 shows the memory requirements of different SNN models under different levels of weight precision. These results show that weight quantization in our lpSpikeCon methodology can significantly decrease the memory footprint, since fewer bits are required to represent all weight parameters of an SNN model. For instance, a model with 8-bit weights reduces the weight memory by 4x (as shown by label H), while a model with 4-bit weights reduces the weight memory by 8x (as shown by label I), as compared to the non-quantized model which employs 32-bit weights. This weight quantization also reduces the memory access requirements, thereby decreasing the number of (off-chip and on-chip) memory accesses. For instance, a non-quantized model requires a single DRAM-based off-chip memory access for obtaining a 32-bit weight, while a quantized model with 8-bit weight can access four weights from a single 32-bit DRAM access [PHS20, PHS21b]. This reduction of memory access requirements is important to enable energy-efficient SNN systems mainly for two reasons. First, memory accesses typically dominate the energy consumption of SNN systems, i.e., about 50%-75% of the total energy consumption of an SNN accelerator [KSVR19]. Second, online training with unsupervised continual learning requires frequent data accesses to memory for updating the weight values at run time. Therefore, reduction of memory access requirements can significantly save the overall system energy, which is especially beneficial for memory- and energy-constrained autonomous agents/systems.

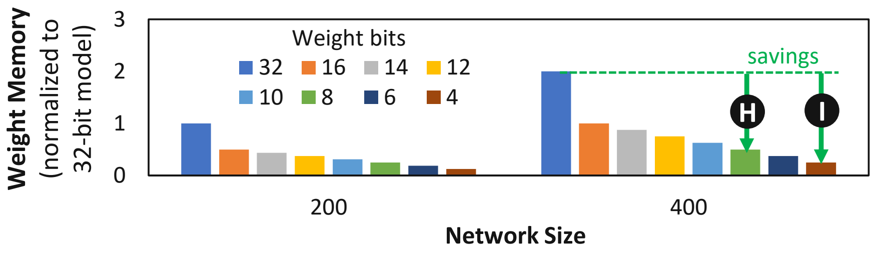


Figure 4.63: Weight memory requirements of SNN models with different sizes of the network (i.e., 200 and 400 excitatory neurons) and different weight precision, which are normalized to the non-quantized SNN model (i.e., 32-bit weights).

The above results and discussion highlight that our lpSpikeCon methodology provides several benefits as compared to the baseline [PS21b], including better learning quality under dynamic environments and lower weight precision, a smaller memory footprint, and higher energy efficiency. Furthermore, our lpSpikeCon can be extended further by incorporating network-specific parameters that have significant impacts on the accuracy. For instance, networks with multiple layers may have additional parameters that should be considered for better adjustments toward adapting to new/unseen features.

### 4.6.7 Summary of lpSpikeCon Methodology

We propose a novel lpSpikeCon methodology to enable low-precision SNN processing for efficient unsupervised continual learning under tight memory budgets (e.g., autonomous agents/systems), through three key steps: (1) analysis of the SNN accuracy profiles, (2) identification of SNN parameters and their adjustment rules, and (3) refinements of parameter values for learning process. As result, our lpSpikeCon significantly reduces the weight memory of an SNN model, while maintaining accuracy in both dynamic and non-dynamic scenarios, as compared to the non-quantized model. Therefore, our lpSpikeCon methodology may enable memory- and energy-efficient autonomous agents/systems that are adaptive to diverse operational environments.

## 4.7 Summary of Energy-Efficient SNN Systems

This chapter discusses our novel methodology for enabling energy-efficient SNN systems. It systematically addresses the targeted problems using our proposed HW/SW-level design and optimization techniques. Specifically, it aims at optimizing the memory and power/energy requirements for SNN processing in both training and inference phases through the reduction of SNN operations, exploration of different quantization techniques for multiple SNN parameters, and employment of approximate DRAM to substantially reduce the dominating power/energy consumption on SNN accelerators while mitigating the negative impact of approximation errors. Furthermore, our proposed techniques also enhance the SNN capabilities to perform unsupervised continual learning for the online learning process through the enhancements of learning rate and the adjustments of multiple SNN parameters (e.g., weight decay and neurons' threshold potential). In this manner, the SNN systems can perform training and inference in tight memory and energy budgets, including their online learning process to continually adapt to diverse operational environments, thereby making such systems suitable for many resource-constrained AI applications (e.g., Edge-AI and Smart CPS). Besides energy efficiency, the SNN systems also need to have reliable processing in the presence of faults. Toward this, Chapter 5 discusses our novel methodology to enable fault-tolerant SNN systems and the respective findings, thereby jointly enabling energy-efficient and fault-tolerant SNN systems.

CHAPTER 5

# Fault-Tolerant SNN Systems

This chapter discusses our novel methodology for achieving fault-tolerant SNN systems. This chapter first identifies the problems toward enabling fault-tolerant SNN inference systems in Section 5.1. Then, to systematically address the research problems, we propose a novel methodology that employs our proposed HW/SW-level fault mitigation techniques for fault-tolerant SNN systems. Specifically, the proposed design flow addresses HW-level approximation-induced errors, permanent faults, and transient faults (i.e., soft errors) as shown in Figure 5.1, and the details of novel contributions are described in the following sections in this chapter. Section 5.2 discusses a framework for mitigating approximation-induced errors and permanent faults in the off-chip and on-chip memories of neuromorphic accelerators through fault-aware training and mapping. Section 5.3 discusses a methodology for mitigating permanent faults in the compute engine of neuromorphic accelerators through fault-aware mapping and the corresponding hardware enhancements without costly retraining. Furthermore, Section 5.4 discusses a methodology for mitigating soft errors that occur in the compute engine of neuromorphic accelerators through weight bounding, neuron protection, and the corresponding hardware enhancements.

## 5.1 Problem Identification

Current trends show that large-sized SNN models are more favorable than the smaller ones since they usually can achieve higher accuracy, but at the cost of higher memory footprint and energy consumption [PS20], as illustrated in Figure 4.2. To address these challenges, neuromorphic accelerators have been developed to improve the performance and energy efficiency of SNN-based applications [BDFZ22]. However, *these neuromorphic accelerators may suffer from accuracy degradation when SNN processing is performed under the presence of HW-induced faults in the off-chip and on-chip memories*; see Ⓐ and Ⓑ in Figure 5.2. The reason is that, faulty memories can alter the values of stored data (e.g., through bit flips), thereby providing incorrect values for SNN computation and leading to incorrect outputs.

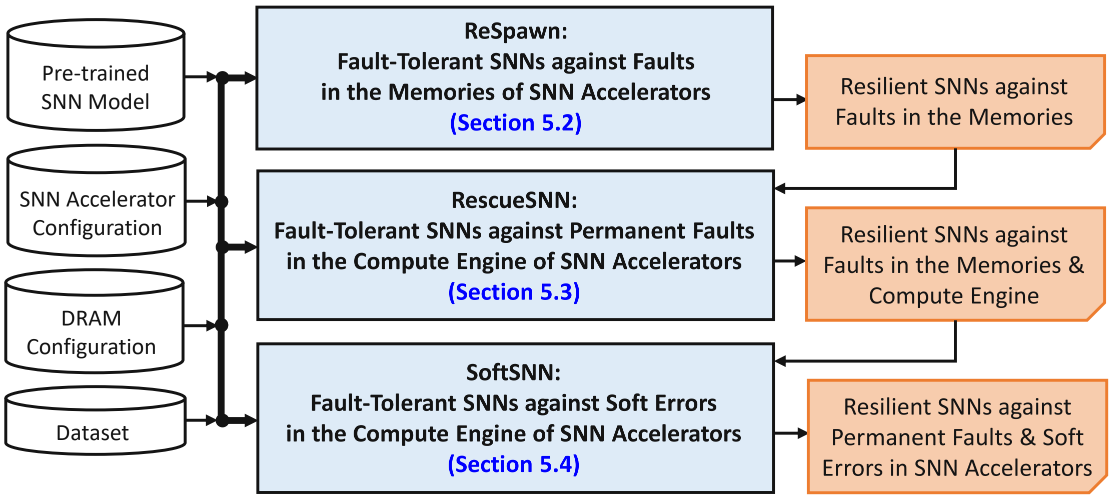

Pre-trained SNN Model
SNN Accelerator Configuration
DRAM Configuration
Dataset
ReSpawn: Fault-Tolerant SNNs against Faults in the Memories of SNN Accelerators (Section 5.2)
RescueSNN: Fault-Tolerant SNNs against Permanent Faults in the Compute Engine of SNN Accelerators (Section 5.3)
SoftSNN: Fault-Tolerant SNNs against Soft Errors in the Compute Engine of SNN Accelerators (Section 5.4)
Resilient SNNs against Faults in the Memories
Resilient SNNs against Faults in the Memories & Compute Engine
Resilient SNNs against Permanent Faults & Soft Errors in SNN Accelerators

Figure 5.1: An overview of the design flow of this chapter.

These faults may come from different sources, as explained in Section 2.4.1 and briefly described in the following.

- *Manufacturing defects*: The imperfections in the chip fabrication process can cause defects in memory cells, hence degrading their functionality. Chips that contain faults are typically discarded/unused, thereby reducing the yield of chips [KK98].

- *Voltage-induced approximation errors*: The operational voltage of memories can be reduced to decrease the power and energy consumption, at the cost of increased fault rates [GKTB15, CYG$^+$17a].

The state-of-the-art works mainly proposed training-based fault-tolerance strategies without considering the underlying SNN HW architectures. For instance, techniques in [SESA$^+$21] employed training with dropouts, neuron saturation detection, and TMR; while techniques in [RLIS21] employed retraining with additional astrocyte units. Therefore, *the impact of bit-level faults in the off-chip and on-chip memories of neuromorphic accelerators on the accuracy, as well as the respective fault mitigation techniques, are still unexplored.* Moreover, *the existing fault mitigation techniques still rely on the costly additional components and retraining process.*

***Problem-1:*** *How can we efficiently mitigate bit-level faults in the off-chip and on-chip memories of neuromorphic accelerators with minimum overheads.*

*Permanent faults can also affect the functionality of the compute engine of neuromorphic accelerators, including the local weight registers (synapses) and neurons*; see C and D in Figure 5.2(a) and Figure 5.2(b). The reason is that, faulty synapses can corrupt the weight values while faulty neurons can deteriorate the neuron behavior (i.e., membrane potential dynamics and spike generation), thereby degrading the accuracy. Permanent faults can

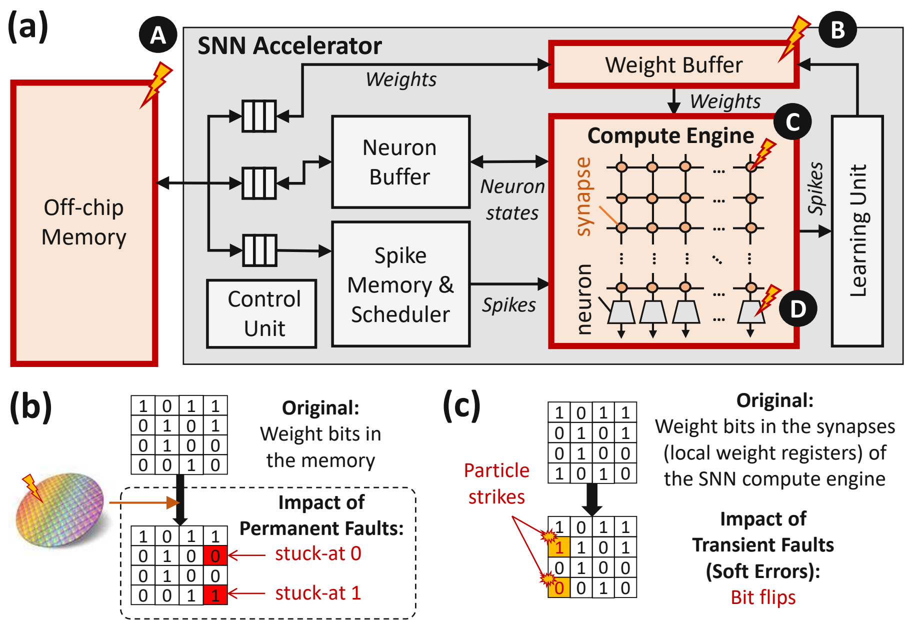


Figure 5.2: (a) The neuromorphic accelerator with faults in the off-chip and on-chip memories: DRAM and weight buffer, as well as faults in the compute engine: synapses (local weight registers) and neurons. (b) Permanent faults may exist in the form of stuck-at 0 and stuck-at 1 faults. (c) High-energy particle strikes trigger soft errors as bit flips in the HW layer, and result in incorrect output in the application layer.

come from different sources as explained in Section 2.4.1, i.e., *manufacturing defects* during the chip fabrication process and *transistors' wear out and damages* during the run-time operation. These faults manifest as stuck-at 0 or stuck-at 1 at the HW layer (e.g., SNN compute engine), and can propagate to the application layer, resulting in incorrect outputs (e.g., misclassification). Simply discarding the faulty chips at design time will lead to low yield and increased per-unit cost of non-faulty chips, while stopping the executions on faulty chips at run time will lead to a short operational lifetime. Therefore, alternate low-cost solutions for mitigating permanent faults in the SNN compute engine are required. Mitigating permanent faults in the compute engine is important as it may significantly improve the reliability of SNNs, considering that this engine is responsible for computing all SNN parameters which dominantly affects the outputs of processing.

The state-of-the-art works proposed fault-aware training with the support of neuron saturation detection and TMR [SESA$^+$21], and additional astrocyte units [RLIS21], without considering the underlying SNN HW architectures. Therefore, *the impact of permanent faults in the SNN compute engine on the accuracy, and the respective low-cost fault mitigation techniques, are still unexplored.*

***Problem-2:*** *How can we efficiently mitigate permanent faults in the compute engine of neuromorphic accelerators with minimum overheads.*

*The neuromorphic accelerators may also suffer from accuracy degradation when SNN processing is performed under the presence of transient faults (i.e., soft errors) in the SNN compute engine, including the local weight registers (synapses) and neurons*; see Ⓒ and Ⓓ in Figure 5.2(a) and Figure 5.2(c). These soft errors occur due to high-energy particle strikes which can come from cosmic rays or packaging materials [Bau05], as discussed in Section 2.4.1. These errors manifest as bit flips at the HW layer, and can propagate to the application layer, resulting in incorrect outputs [Bau05]. Soft errors in the local weight registers (synapses) and the neurons can affect the functionality of the SNN compute engine, e.g., by corrupting the weight values and the behavior of neuron operations including operations for membrane potential dynamics and spike generation. Therefore, alternate low-cost solutions for mitigating soft errors in the SNN compute engine are required. Mitigating soft errors in the compute engine is important as it may significantly improve the reliability of SNNs, considering that this engine dominantly affects the outputs of processing. However, state-of-the-art works have not studied the SNN fault tolerance considering soft errors in the underlying hardware. Therefore, *the impact of soft errors in the compute engine (i.e., local weight registers and neurons) on the accuracy, and the respective lightweight mitigation techniques are still unexplored.*

***Problem-3:*** *How can we efficiently mitigate soft errors in the compute engine of neuromorphic accelerators with minimum overheads.*

**Benefits:** The solution to these problems will enable *reliable SNN processing even in the presence of permanent and transient faults in the memories and compute engine of neuromorphic accelerators for energy-constrained embedded platforms and their applications for Edge-AI and Smart CPS*. The solution will also enable wafer-scale chips for SNNs where embracing permanent faults is important to maintain the yield, and reduce the per-unit cost of the neuromorphic chips.

**Proposed Solution:** To systematically address the above problems, we propose a comprehensive solution which is discussed in several sections. Specifically, Problem-1, Problem-2, and Problem-3 are addressed in Section 5.2, Section 5.3, and Section 5.4, respectively.

## 5.2 ReSpawn: Energy-Efficient Fault-Tolerance for SNNs considering Unreliable Memories

This section aims at addressing **Problem-1** with the solution for efficiently mitigating bit-level faults in the off-chip and on-chip memories of the SNN neuromorphic accelerators.

### 5.2.1 Motivational Study

To understand the impact of faults in the memories of neuromorphic accelerators on the accuracy, we perform experiments that explore different fault rates in the DRAM-based

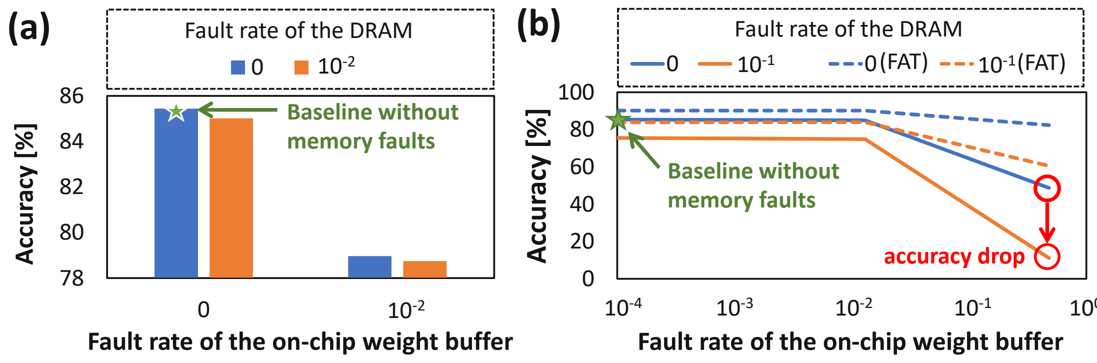


Figure 5.3: (a) Impact of faults in the DRAM and the weight buffer on the accuracy. (b) Increasing fault rates in weight memories lead to accuracy degradation, and fault-aware training (FAT) can improve the SNN fault tolerance.

off-chip memory and the SRAM-based on-chip weight memory (i.e., weight buffer), while considering the typical architecture of neuromorphic accelerators shown in Figure 2.14. Here, we consider 256x256 synapses with 8-bit precision of weights, 256 neurons, a 2Gb DDR3-1600 DRAM, a 32 KB weight buffer, and a uniform random distribution of faults on each bank of the DRAM and the weight buffer in the form of bit flips. For the network, we consider the FC-based SNN in Figure 4.8(a) with 900 excitatory neurons. Further details on the experimental setup are presented in Section 5.2.6. The experimental results are presented in Figure 5.3, from which we make the following key observations.

- Different fault rates in the DRAM and the weight buffer cause an SNN system to obtain different accuracy scores. Higher fault rates in the DRAM and the weight buffer typically lead to lower accuracy.
- Faults in the weight buffer have a relatively higher impact on the accuracy degradation than the DRAM since its size is significantly smaller than the DRAM, and thereby having a higher probability to affect more weights, as shown in Figure 5.3(a).
- Fault-aware training (FAT) techniques with progressive fault injection for neural networks [KOY$^+$19, PHS21c] can improve the SNN fault tolerance while incurring high training time and energy consumption, as such techniques considers a wide range of fault rates for the injection.

### 5.2.2 Scientific Research Challenges

The above observations expose key challenges that need to be solved for addressing the targeted research problem, as discussed in the following.

- *The fault-mitigation technique should minimize the impacts of faults in both, the DRAM and the weight buffer*, thereby improving the SNN fault tolerance.
- *It should employ a technique that does not rely on retraining*, as retraining needs a full training dataset, which may not be available due to restriction policies (e.g., a company releases an SNN model, but makes the training dataset unavailable).

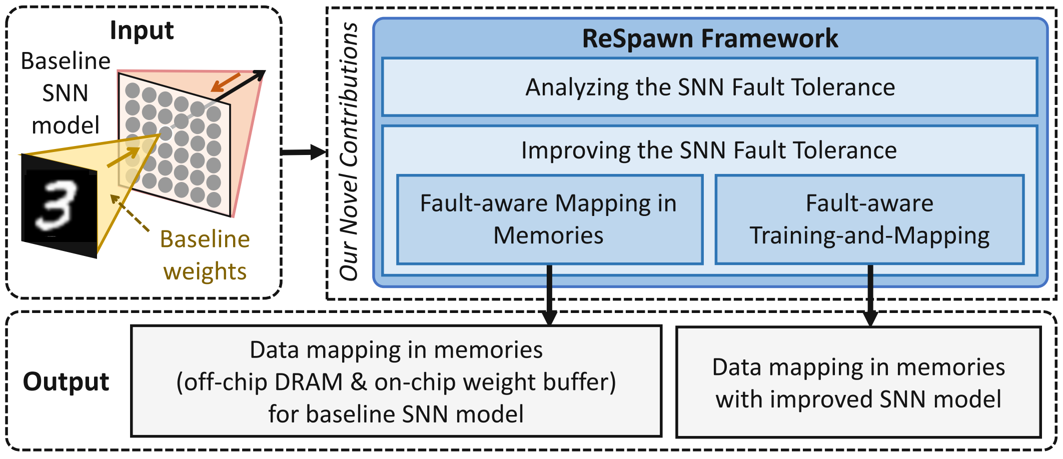
- *It should incur low energy overhead at run time*, as compared to the baseline (without fault-mitigation technique) to enable energy-efficient SNN applications.

***Required:*** *A low-cost technique for mitigating the negative impact of faults in the off-chip and on-chip weight memories of neuromorphic accelerators, thus enabling reliable SNN inference in an energy-efficient manner.*

### 5.2.3 Novel Contributions

To address the above challenges, we propose ReSpawn, *a novel framework that enables energy-efficient fault-toleRance for Spiking neural networks considering unreliable memories*. To the best of our knowledge, this work is the first effort that mitigates the negative impacts of faults in the off-chip and on-chip weight memories of neuromorphic accelerators. Following are the key steps of the ReSpawn framework, and its overview is shown in Figure 5.4.

1. **Analyzing the fault tolerance of the SNN model** to characterize the accuracy values under the given fault rates. It is performed by adjusting the fault rates in the memories, while checking the obtained accuracy.
2. **Improving the fault tolerance of the SNN model** whose strategies depend on the availability of the training dataset.
   - *If the training dataset is not fully available, then the Fault-aware Mapping (FAM)* is employed through simple bit-shuffling techniques, that prioritize placing the bits with higher significance in the non-faulty memory cells.
   - *If the training dataset is fully available, then the Fault-aware Training-and-Mapping (FATM)* is employed by including the information of the faulty memory cells in the data mapping and training processes. Here, the data mapping strategy follows the proposed FAM technique.


Input
Baseline SNN model
Baseline weights
Our Novel Contributions
ReSpawn Framework
Analyzing the SNN Fault Tolerance
Improving the SNN Fault Tolerance
Fault-aware Mapping in Memories
Fault-aware Training-and-Mapping
Output
Data mapping in memories (off-chip DRAM & on-chip weight buffer) for baseline SNN model
Data mapping in memories with improved SNN model
Figure 5.4: An overview of our novel contributions, which are shown in blue boxes.

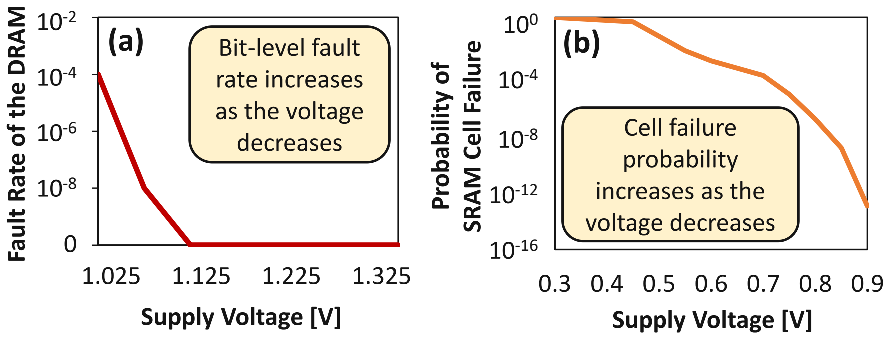


Figure 5.5: (a) DRAM fault rates and the corresponding DRAM voltage values, based on the study in [CYG+17a]. (b) SRAM cell failure probability ($P_{cell}$) and the corresponding SRAM voltage values for a 28 nm CMOS technology, based on the study in [GKTB15]. The yield of non-faulty cells is defined as $Y = (1 - P_{cell})^M$ with $M$ denotes the total memory bit-cells.

### 5.2.4 Fault Modeling for Memories

We focus on the fault modeling for the DRAM and the SRAM weight buffer, since we aim to accurately explore the impacts of hardware-induced faults in weight memories across the hierarchy of a neuromorphic accelerator, as indicated by Ⓐ and Ⓑ in Figure 5.2. These faults can come from *manufacturing defects* due to the imperfections in the fabrication process [KK98, THG17, HKP+18, ZLK+19, SNT+20b], and *reduced-voltage operation* which is performed for decreasing the operational power/energy [CYG+17a, GKTB15].

**Faults from Manufacturing Defects:** The neuromorphic HW accelerators are fabricated using a sophisticated manufacturing process. Therefore, there is a chance of imperfections that result in defects in the fabricated chips. Moreover, the technology scaling (which is employed for improving the performance and efficiency of the chips) may increase fault rates related to permanent faults at random locations of a chip. Therefore, the faults from manufacturing defects can be modeled using a uniform random distribution and can manifest in the form of bit flips, which has also been considered in previous works [ZGBG18, HS20].

**Faults from Reduced-Voltage Operations:** The reduction of operational voltage is a widely-used approach to reduce the operational power/energy of DRAM and SRAM-based buffer, at the cost of increased fault rates, as shown in Figure 5.5. For DRAM, we follow the fault model from [KOY+19], i.e., the faults are modeled by considering the *weak cells* (i.e., cells that fail when the DRAM voltage is reduced), and the probability of a fault in any weak cell. These faults typically have a uniform random distribution across a DRAM bank and manifest in the form of bit flips. Meanwhile, for SRAM, we follow the fault model from [GKTB15], i.e., the faults have a uniform random distribution across an SRAM bank. The selection of the uniform random distribution as the fault model for DRAM and SRAM, is motivated by the following reasons: (1) it produces faults with high similarity to the real reduced-voltage DRAM [KOY+19] and the real reduced-voltage SRAM [GKTB15]; and (2) it offers fast software-based fault injection.

### 5.2.5 ReSpawn Framework

We propose the ReSpawn framework [PHS21a] to enable energy-efficient fault-tolerance for SNN inference on unreliable off-chip and on-chip weight memories. The key steps of our ReSpawn are shown in Figure 5.6 and discussed in the following sections.

1. **Analyzing the SNN fault tolerance (Section 5.2.5.1):** It aims at understanding the interaction between the fault rates and the accuracy, by exploring different combinations of fault rates in DRAM and weight buffer, while observing the accuracy scores. This information is then leveraged for improving the SNN fault tolerance.

2. **Improving the SNN fault tolerance** through different strategies, depending on the availability of the training dataset.

   - **Fault-aware Mapping (Section 5.2.5.2):** *This strategy is performed if the training dataset is not fully available.* It employs efficient bit-shuffling techniques, that map the significant bits to the non-faulty memory cells and the insignificant bits to the faulty ones. We propose two FAM techniques to offer accuracy-energy trade-offs.
     - **FAM1:** It considers the fault map from each memory as an individual fault map, and devises a mapping pattern for each fault map.
     - **FAM2:** It merges multiple fault maps from off-chip and on-chip memories to an integrated fault map, and devises a mapping pattern for it accordingly.
   - **Fault-aware Training-and-Mapping (Section 5.2.5.3):** *This strategy is performed if the training dataset is fully available.* It uses the information of the faulty memory cells in the data mapping and training processes. Here, the data mapping strategy also follows the proposed FAM techniques (i.e., FAM1 and FAM2).

#### 5.2.5.1 SNN Fault Tolerance Analysis

Understanding the fault tolerance of the given SNN model is important, because the information from the analysis will be beneficial, especially for performing efficient fault-mitigation techniques. Therefore, *our ReSpawn framework analyzes the fault tolerance of the SNN model to observe the interaction between memory faults and accuracy.* It is performed by exploring different combinations of fault rates in the DRAM and the weight buffer, while observing the obtained accuracy. For instance, if we consider a network with 900 neurons, our ReSpawn will explore different combinations of fault rates in DRAM and weight buffer, and the experimental results are shown in Figure 5.7. These results show two different regions, i.e., where fault rates in memories cause the network to achieve acceptable accuracy, as shown by label-Ⓐ, and where fault rates in memories cause the network to suffer from notable accuracy degradation, as shown by label-Ⓑ. These regions provide insights regarding the tolerable fault rates that should be considered to effectively improve the SNN fault tolerance.

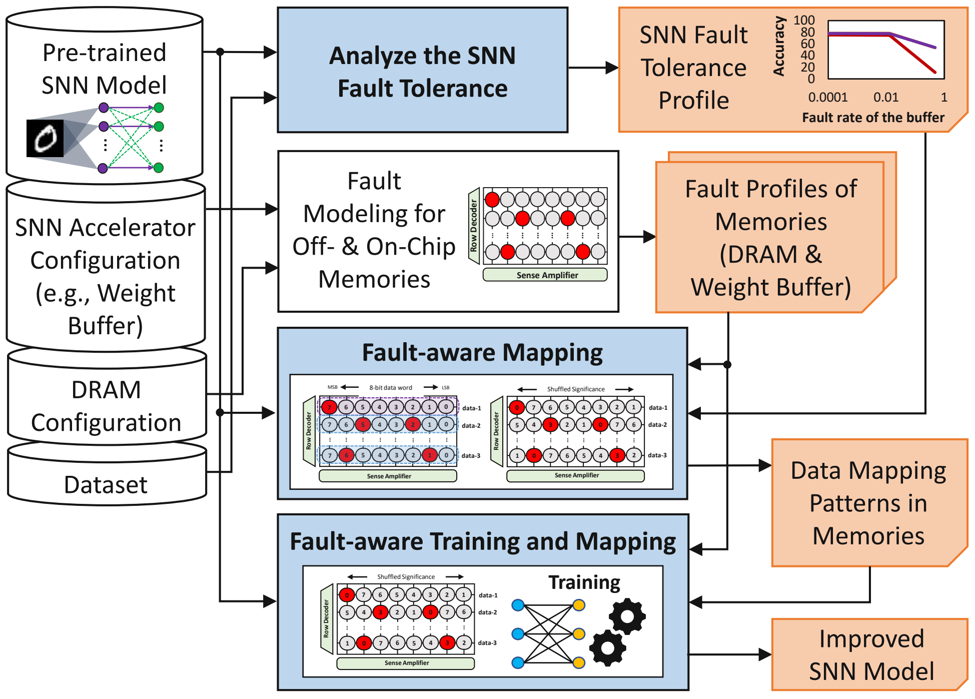

Pre-trained SNN Model
SNN Accelerator Configuration (e.g., Weight Buffer)
DRAM Configuration
Dataset
Analyze the SNN Fault Tolerance
SNN Fault Tolerance Profile
Accuracy
Fault rate of the buffer
Fault Modeling for Off- & On-Chip Memories
Row Decoder
Sense Amplifier
Fault Profiles of Memories (DRAM & Weight Buffer)
Fault-aware Mapping
8-bit data word
Shuffled Significance
Data Mapping Patterns in Memories
Fault-aware Training and Mapping
Training
Improved SNN Model

Figure 5.6: An overview of the ReSpawn framework. The novel contributions are highlighted in blue.

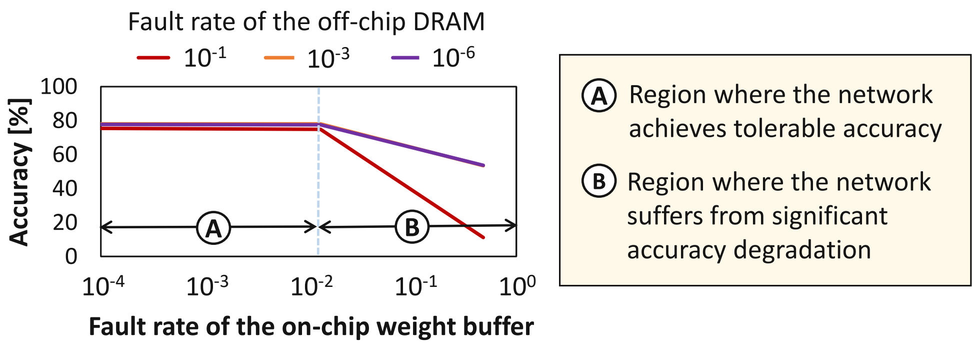


Figure 5.7: The experimental results for a 900-neuron network, considering different fault rates for DRAM and weight buffer.

### 5.2.5.2 Fault-aware Mapping (FAM)

Faulty cells in memories that come from manufacturing defects and reduced-voltage operations, can be characterized at design time. Therefore, their locations are known before the deployment. *Our ReSpawn leverages the information of faulty cells in the DRAM and the weight buffer to effectively map the weights to memory fabrics, thereby minimizing the impact of faulty cells on the significant bits.* It is performed through FAM that employs simple bit-shuffling techniques for placing the significant bits in the non-faulty memory cells and the insignificant bits in the faulty ones.

Furthermore, we observe that, a data word may have a single faulty bit or multiple faulty bits, depending on whether this word occupies a memory segment that has a single faulty cell or multiple faulty cells, as illustrated in Figure 5.8(a). Therefore, we propose a mapping strategy that can address both, the single fault-per-word and multiple faults-per-word scenarios.

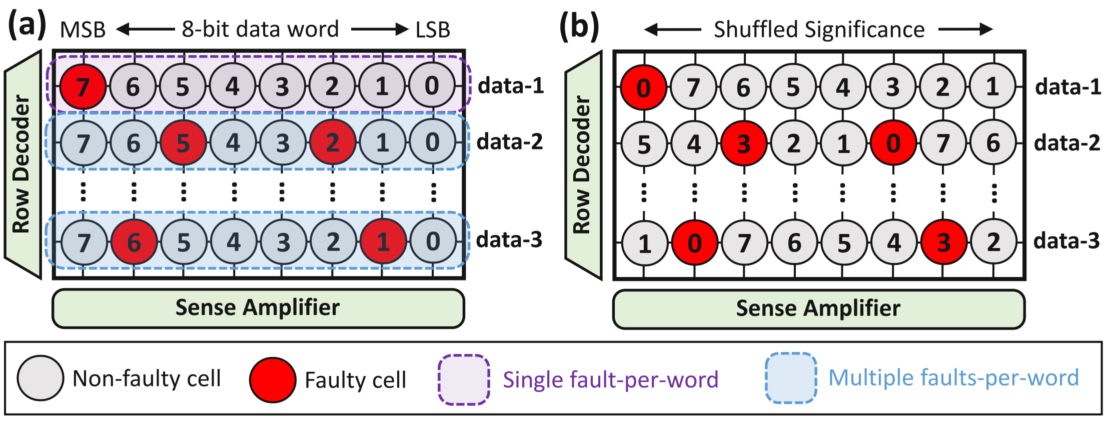


Figure 5.8: (a) Illustration of possible locations of faulty cells in memories. (b) The proposed bit-shuffling technique, which is based on the right circular shift.

The proposed memory mapping strategy is performed by the following steps, which are also illustrated in Figure 5.8(b).

- **Step-1:** *Identifying the faulty cells in the given memories.* This step aims at obtaining information regarding faulty cells in each memory, such as fault rate and fault map. The faulty cells in the on-chip buffer from manufacturing defects can be detected using the standard post-fabrication testing [ZGBG18], and the faulty cells in the DRAM can be detected through measurements, e.g., using SoftMC tool [HVK<sup>+</sup>17]. Meanwhile, the faulty cells from reduced-voltage operations can also be detected through measurements on the DRAM [HVK$^+$17] and on the on-chip buffer [GKTB15]. In this manner, collecting the faulty cell information is feasible as it follows the standard post-fabrication testing and measurements.

- **Step-2:** *Identifying the maximum fault rate allowed in a data word.* This step aims at determining which memory cells can be used for storing a data word, by considering fault rates and accuracy from the SNN fault tolerance analysis in Section 5.2.5.1. For instance, we allow a maximum of 2 faulty bits for an 8-bit data word.

- **Step-3:** *Identifying the memory segment with the highest number of subsequent non-faulty cells for storing a data word.* It aims at maximizing the possibility of placing the significant bits in the non-faulty cells. Therefore, we also examine the corner case (i.e., the left-most memory cell with the right-most memory cell) as possible subsequent non-faulty cells, as shown in the second row of Figure 5.8(b).

- **Step-4:** *Performing circular-shift technique for each data word.* It efficiently performs bit-shuffling by employing the right circular shift, and hence simplifying the control.

Since the FAM technique leverages the information of fault maps from multiple memories, we propose two variants of FAM techniques to offer different accuracy-energy trade-offs, which are discussed in the following.

### FAM for Individual Fault Map (FAM1)

This technique considers an individual fault map from each memory (i.e., DRAM or weight buffer). Therefore, the FAM1 devises multiple mapping patterns, i.e., one pattern for DRAM, and another one for weight buffer, as illustrated in Figure 5.9. This FAM1 technique offers high resiliency against faults from each memory, as each mapping pattern minimizes the negative effect of faults on the significant bits. However, it needs to perform a specialized data mapping for each memory.

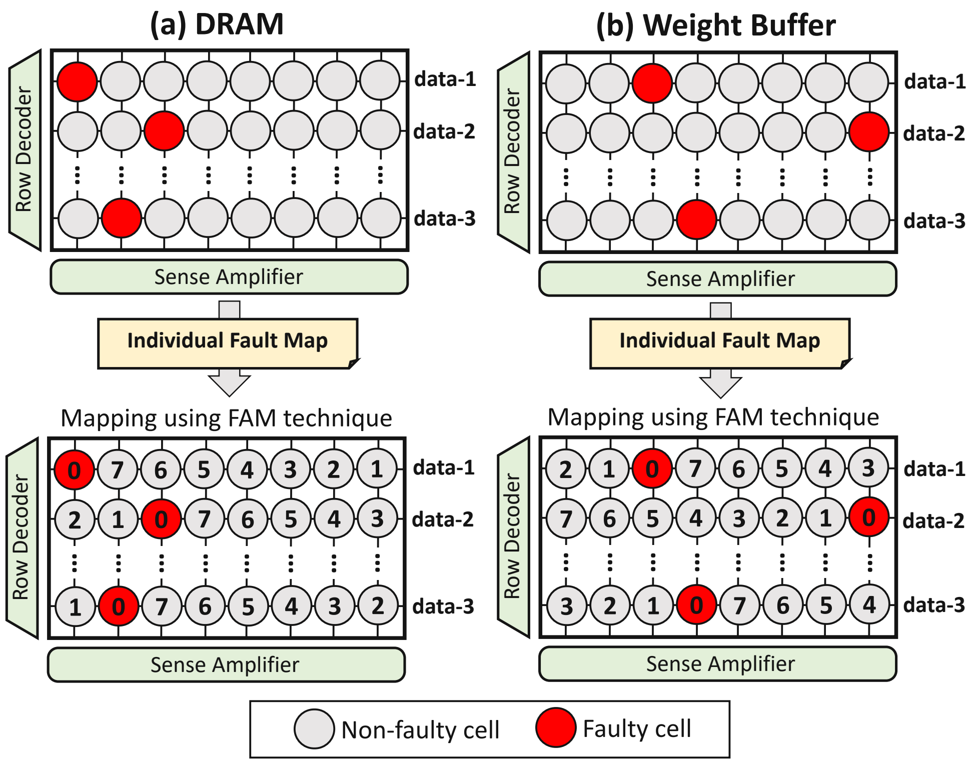


Figure 5.9: (a) For DRAM, the FAM1 only considers the DRAMs' fault map. (b) For the weight buffer, the FAM1 only considers the buffers' fault map.

### FAM for Integrated Fault Map (FAM2)

This technique merges multiple fault maps from multiple memories (i.e., DRAM and weight buffer) as an integrated fault map. Therefore, the FAM2 only devises a single mapping pattern for both, DRAM and weight buffer, as illustrated in Figure 5.10. This FAM2 technique potentially offers better efficiency than the FAM1, due to its simpler mapping technique. However, it is less resilience than FAM1 as the generated mapping pattern may be sub-optimal for each memory, because some insignificant bits may be placed in non-faulty cells and some significant bits in faulty ones.

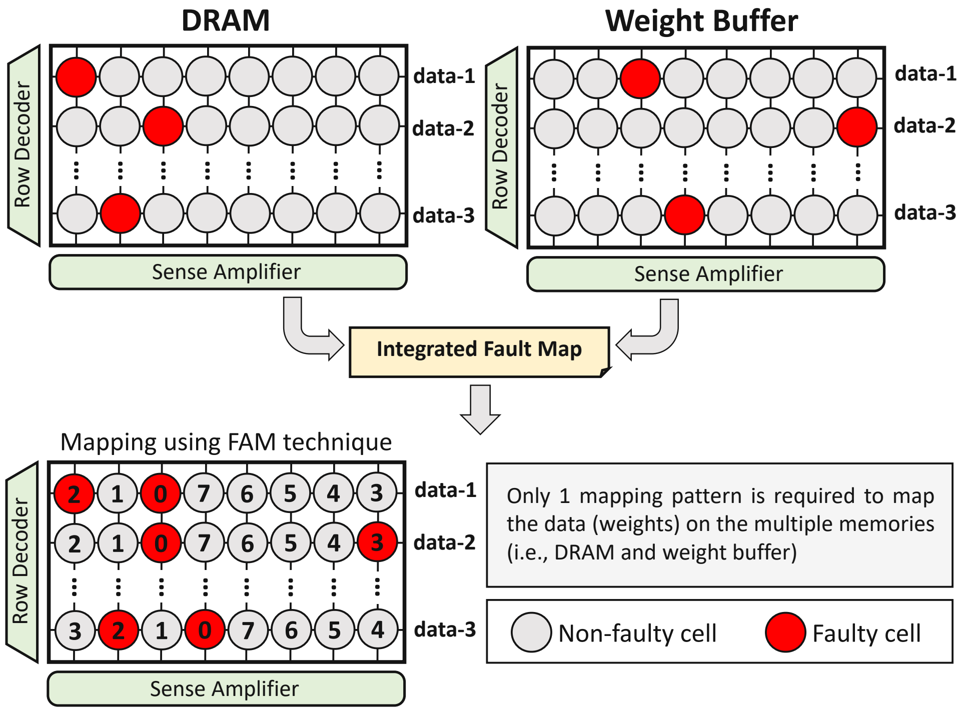

DRAM
Weight Buffer
Row Decoder
data-1
data-2
data-3
Sense Amplifier
Integrated Fault Map
Mapping using FAM technique
Only 1 mapping pattern is required to map the data (weights) on the multiple memories (i.e., DRAM and weight buffer)
Non-faulty cell
Faulty cell

Figure 5.10: The FAM2 technique considers the integrated fault map for devising the mapping pattern for both, DRAM and weight buffer.

ReSpawn also considers optimizing the energy of DRAM and SRAM buffer accesses to maximize the energy efficiency potential, since memory accesses typically dominate the total energy of SNN processing [KSVR19]. The DRAM mapping is performed by maximizing the DRAM row buffer hits [GLH+19], multi-bank burst feature [PHS21b], and subarray-level parallelism [KSL+12, PHS20], while considering the proposed FAM techniques (the algorithm is presented in Algorithm 13). Meanwhile, the SRAM buffer mapping is performed by maximizing the bank-level parallelism [PHS21b] while considering the proposed FAM techniques (the algorithm is presented in Algorithm 14).

Note that the proposed FAM techniques (i.e., FAM1 and FAM2) do not require retraining, thereby making them suitable for energy-efficient and fault-tolerant SNN processing, especially in the case where the training dataset is not fully available. Consequently, these techniques can also improve the yield and reduce the per-unit cost of neuromorphic accelerators/chips.

**Algorithm 13** The proposed mapping for a DRAM chip

```
INPUT: (1) DRAM ($DRAM$), number of bank-per-chip ($D_{ba}$), number of subarray-per-bank
    ($D_{su}$), number of row-per-subarray ($D_{ro}$), number of column-per-row ($D_{co}$);
    (2) Fault rate of a DRAM column ($Drate\_col$), maximum tolerable fault rate for a DRAM
    column ($Drate\_col_{max}$);
    (3) Weight bits ($weight\_b$);
    (4) Fault-aware mapping ($FAM$); // either FAM1 or FAM2
OUTPUT: DRAM ($DRAM$);
BEGIN
    Process:
1:  for $ro = 0$ to $(D_{ro} - 1)$ do
2:      for $su = 0$ to $(D_{su} - 1)$ do
3:          for $ba = 0$ to $(D_{ba} - 1)$ do
4:              for $co = 0$ to $(D_{co} - 1)$ do
5:                  if $Drate\_col \leq Drate\_col_{max}$ then
6:                      $DRAM[ba, su, ro, co] \leftarrow FAM(weight\_b)$;
7:  return $DRAM$;
END
```

**Algorithm 14** The proposed mapping for SRAM buffer

```
INPUT: (1) SRAM ($SRAM$), number of bank ($S_{ba}$), number of row-per- bank ($S_{ro}$); // the
    number of column-per-row = the bitwidth of a word
    (2) Fault rate of an SRAM row ($Srate\_row$), maximum tolerable fault rate for an SRAM
    row ($Srate\_row_{max}$);
    (3) Weight bits ($weight\_b$);
    (4) Fault-aware mapping ($FAM$); // either FAM1 or FAM2
OUTPUT: SRAM ($SRAM$);
BEGIN
    Process:
1:  for $ro = 0$ to $(S_{ro} - 1)$ do
2:      for $ba = 0$ to $(S_{ba} - 1)$ do
3:          if $Srate\_row \leq Srate\_row_{max}$ then
4:              $SRAM[ba, ro] \leftarrow FAM(weight\_b)$;
5:  return $SRAM$;
END
```

### 5.2.5.3 Fault-aware Training-and-Mapping (FATM)

If the training dataset is fully available, users can decide if they want to perform fault mitigation without training, like our FAM techniques (Section 5.2.5), or fault-aware training (FAT). Note, FAT is a widely used technique for improving the fault-tolerance of neural networks, by incorporating the information of faults in the training process [KOY+19, PHS21c, ZGBG18]. Our experimental results in Figure 5.3(c) show that, the FAT technique can improve the SNN fault tolerance. Toward this, *ReSpawn framework also provides FAT-based solutions to improve the SNN fault tolerance on top of the proposed FAM techniques; so-called fault-aware training-and-mapping (FATM).* For conciseness, the FAT with FAM1 mapping is referred to as the FATM1, and the FAT with FAM2 mapping is referred to as the FATM2. The proposed FATM is performed through the following mechanisms.

1. We employ the FAM techniques (FAM1 or FAM2) on the SNN model, to minimize the negative impacts of faults on the weights. This results in the model whose weights have been minimally affected by the faults (i.e., the FAM-improved SNN model).
2. Then, we perform training on the FAM-improved SNN model through the following.
   - **Step-1:** The faults are generated for different rates, based on the SNN fault tolerance analysis in Section 5.2.5.1.
   - **Step-2:** The generated faults are injected into locations in DRAM and weight buffer, thereby causing the weight bits stored in these locations to flip.
   - **Step-3:** We train the SNN model while considering fault rates that do not cause accuracy drop (fault rates from region-Ⓐ in Figure 5.7) and are close to region-Ⓑ. It makes the model adaptive to high fault rates safely, without causing accuracy to decrease, and with less training time, since smaller fault rates are not considered.
   - **Step-4:** Afterward, we carefully train the SNN model while considering fault rates that cause notable accuracy drop, i.e., fault rates from region-Ⓑ, by incrementally increasing the fault rates of DRAM and weight buffer, after each training epoch.
   - **Step-5:** Training is terminated when the network faces accuracy saturation or degradation. The final SNN model is selected from the trained model that is saved in the previous training epoch.

The proposed FAM (FAM1 and FAM2) and FATM (FATM1 and FATM2) techniques are applicable to different memory technologies (like CMOS, RRAM, etc.) since they consider bit-level fault mitigation, which is suitable for bit-level data storage in each memory cell. Therefore, the possible extension to the ReSpawn framework is by considering the multi-level cell characteristics in its optimization process.

### 5.2.6 Evaluation Methodology

The experimental setup for evaluating ReSpawn framework is illustrated in Figure 5.11. Following is detailed information on the evaluation methodology, comprising the scenarios for experiments and comparisons. For the network architecture, we use the fully-connected SNN, like the network in Figure 4.8(a), with a different number of neurons (i.e., 100, 400, 900, 1600, 2500, and 3600) to show the generality of the ReSpawn, which we refer them to as Net100, Net400, Net900, Net1600, Net2500, and Net3600, respectively. We consider this network as it provides robustness when performing different variants learning rules [DC15], thereby it is representative for the evaluation. Meanwhile, for the comparison partners, we consider two designs: (1) the baseline SNN model without any fault-mitigation technique, and (2) the SNN model with the FAT technique. We compare these designs against our ReSpawn techniques (i.e., FAM1, FAM2, FATM1, and FATM2) on the MNIST dataset.

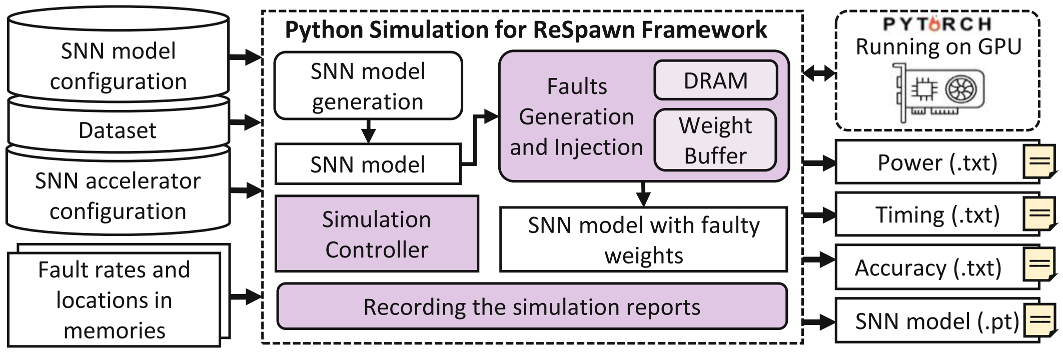

SNN model configuration
Dataset
SNN accelerator configuration
Fault rates and locations in memories
Python Simulation for ReSpawn Framework
SNN model generation
SNN model
Faults Generation and Injection
DRAM
Weight Buffer
Simulation Controller
SNN model with faulty weights
Recording the simulation reports
PYTORCH
Running on GPU
Power (.txt)
Timing (.txt)
Accuracy (.txt)
SNN model (.pt)

Figure 5.11: Experimental setup and tool flow.

**Fault Injection:** We generate memory faults based on the fault modeling described in Section 5.2.4. Afterward, we inject these faults into the locations in DRAM and weight buffer to represent the faulty memory cells, and the data bits in these cells are flipped. For the ReSpawn, we employ mapping policies from Algorithm 13 and Algorithm 14, while for the baseline, we store the weights in the subsequent addresses of a DRAM bank.

**Accuracy Evaluation:** To evaluate the accuracy of SNNs, we use Python-based simulations [HSK$^+$18] that run on GPGPU machine, i.e., Nvidia RTX 2080 Ti, while considering an SNN accelerator architecture that follows the design in Figure 2.14 with 8-bit precision of weights, a DDR3-1600 2Gb DRAM, and a 32KB weight buffer. We use 8-bit precision as it has a sufficient range of values to represent the SNN weights [PS20].

**Energy Evaluation:** We consider the approach in [HMD16] for estimating the SNN processing energy of an SNN model, i.e., by leveraging the information of processing power that is obtained through *nvidia-smi* utility, and its processing time. We perform the energy evaluation for different scenarios, i.e., the fault-mitigation techniques without retraining (i.e., FAM1 and FAM2) and with retraining (i.e., FAT, FATM1, and FATM2).

### 5.2.7 Results and Discussion

#### 5.2.7.1 Mantaining the Accuracy

Figure 5.12 presents the experimental results on the accuracy of different fault-mitigation techniques, i.e., baseline, FAT, our FAM techniques (i.e., FAM1 and FAM2), and our FATM techniques (i.e., FATM1 and FATM2).

We observe that the baseline is susceptible to accuracy degradation when the SNN is run under the presence of faults, as these faults alter the weights and affect the output of the SNN model. The accuracy degradation is more evident in the scenarios where high fault rates are observed, as shown by label-①. The FAT technique improves the SNN fault tolerance compared to the baseline across all evaluation scenarios, as the FAT-improved SNN model has a better capability for adapting to the presence of faults, as shown by label-②. However, the FAT technique may offer limited accuracy improvements since

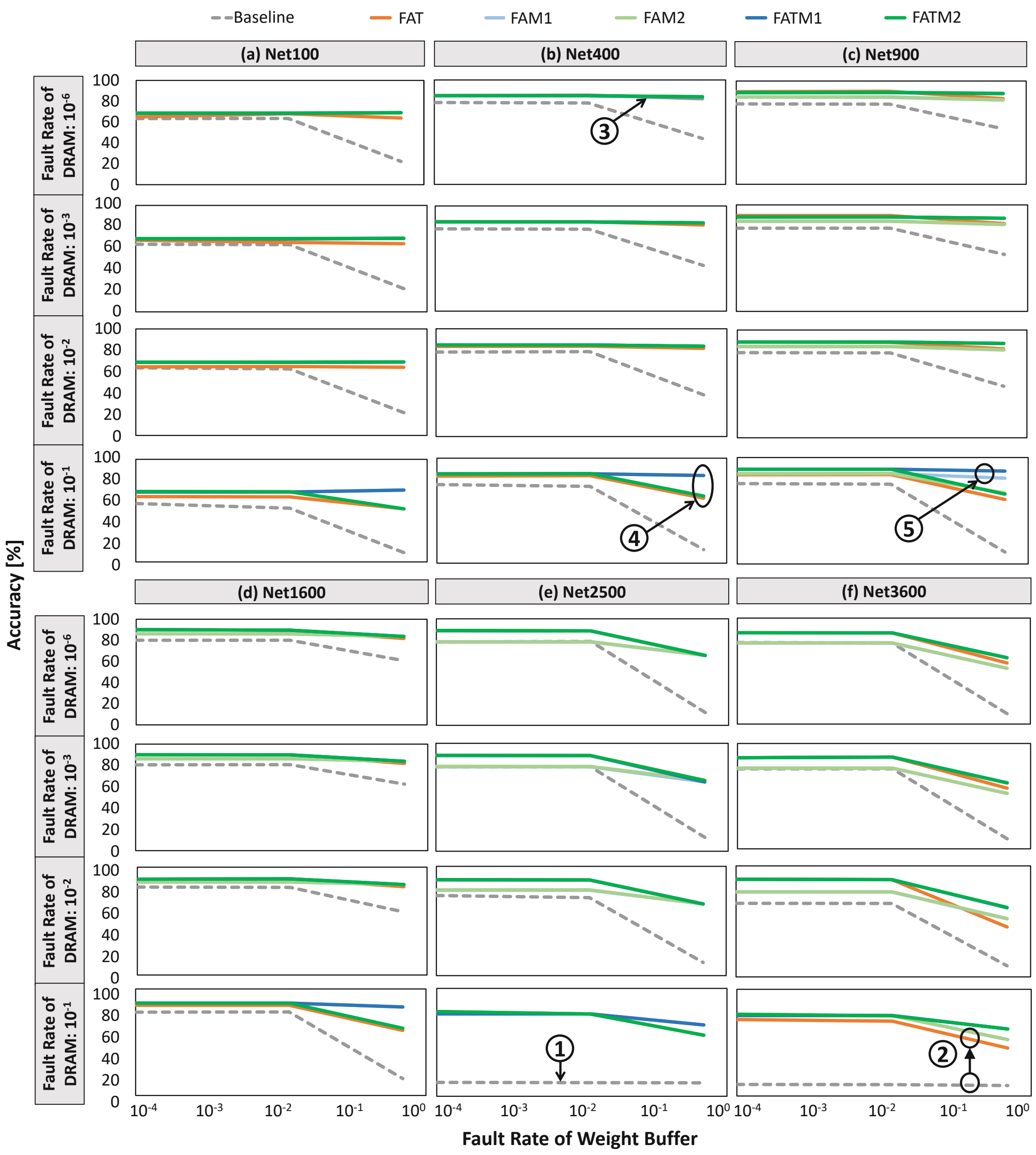


Figure 5.12: Accuracy achieved by different techniques, across different sizes of networks: (a) Net100, (b) Net400, (c) Net900, (d) Net1600, (e) Net2500, and (f) Net3600, as well as different fault rates in DRAM and weight buffer.

it does not substantially eliminate the negative impact of faults on the significant bits of weights, and its performance depends on the effectiveness of the training strategy. On the other hand, our FAM techniques (FAM1 and FAM2) can achieve comparable accuracy compared to the FAT without retraining, as shown by label-③. They improve the accuracy by up to 61%, 70%, 70%, 67%, 53%, and 43% for Net100, Net400, Net900, Net1600, Net2500, and Net3600 respectively, as compared to the baseline. The reason is that, the main idea of our FAM1 and FAM2 is to eliminate the impact of faults on the significant bits of weights through simple bit-shuffling, thereby maintaining the value of weights as close as possible to the weights that are trained in an ideal condition, i.e., environment without faults. These results show that *our FAM techniques (FAM1 and FAM2) are effective for fault-mitigation techniques in SNNs, especially in the case where the training dataset is not fully available.* Moreover, these techniques can enhance the yield, thereby decreasing the per-unit cost of SNN chips.

We also observe that the FAM1 has better performance than the FAM2, as it consistently obtains high accuracy across all evaluation scenarios, while the performance of the FAM2 may still be affected by cases that have high fault rates, as shown by label-④. The reason is that, the FAM1 minimizes the impact of faults on the significant bits of weights from each memory. Meanwhile, the FAM2 minimizes the impact of faults on the significant bits of weights considering the integrated fault map from multiple memories, which may be sub-optimal for each memory. We also observe that our FATM techniques can further improve the SNN fault tolerance from the FAM techniques (shown in label-⑤), since the training is performed on the model whose weights are already minimally affected by the faults. The FATM techniques improve the accuracy by up to 61%, 70%, 76%, 67%, 53%, and 53% for Net100, Net400, Net900, Net1600, Net2500, and Net3600 respectively, as compared to the baseline. These results show that, *our FATM techniques (FATM1 and FATM2) can further improve the SNN fault tolerance if the training dataset is fully available.*

### 5.2.7.2 Reducing the Energy Consumption

Figure 5.13 presents the experimental results on the energy consumption of different fault-mitigation techniques, i.e., the baseline, the FAT, our FAM1, FAM2, FATM1 and FATM2. For the training-based solutions (FAT, FATM1, and FATM2), the energy consumption is evaluated when performing one training epoch over 60K samples from the full MNIST training set. For the solutions without training (FAM1 and FAM2), the energy consumption is evaluated when performing a test over the 10K samples from the full MNIST test set. *This evaluation scenario aims at showing how much energy the training-based solutions incur, as compared to the solutions without training.*

We observe that, the FAT technique consumes high energy across different network sizes, since it requires the training process. Moreover, a larger-sized network incurs higher power and processing time, and thereby higher energy. If we consider one training epoch (i.e., running 60K samples from the full MNIST training set), the FAT incurs about 6x-20x energy for Net100-Net3600, as compared to the baseline. This condition can be exacerbated by the fact that energy consumption is increased if the FAT requires multiple training epochs. Here, the FATM1 and FATM2 techniques face the same issues due to the training-based approach. On the other hand, our FAM1 technique only incurs 1.03x-1.12x energy, and our FAM2 technique only incurs 1.02x-1.09x energy for Net100-Net3600 as compared to the baseline, when running 10K samples from a full MNIST test set.

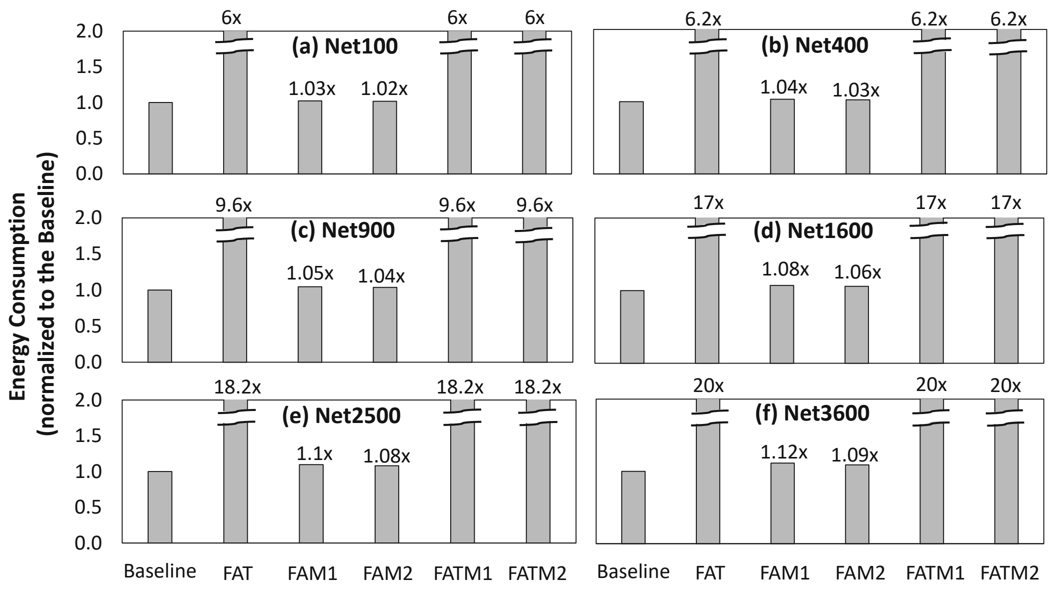


Figure 5.13: Normalized energy consumption of different fault-mitigation techniques (i.e., the baseline, the FAT, our FAM1 and FAM2, as well as our FATM1 and FATM2) across different sizes of network: (a) Net100, (b) Net400, (c) Net900, (d) Net1600, (e) Net2500, and (f) Net3600.

Moreover, the energy efficiency of the FAM1 and the FAM2 can be better if the number of samples in the inference phase is higher. The reason is that, the mapping patterns in the FAM1 and the FAM2 need to be generated only once, before running the inference, therefore the energy overhead is negligible considering the huge number of samples to be processed in the inference phase. We also observe that the FAM2 incurs slightly less energy compared to the FAM1, as the FAM2 only considers one integrated fault map for its mapping operations while the FAM1 considers multiple fault maps for its mapping operations, thereby incurring fewer operations and processing energy. These results show that, *our FAM techniques (FAM1 and FAM2) have high potential as the energy-efficient fault-mitigation techniques for SNNs*, as they maintain high accuracy with minimum energy overhead.

### 5.2.8 Further Discussion

Our ReSpawn methodology leverages the SNN fault-tolerance analysis considering the given fault model, before applying the FAM or FATM technique. Therefore, the impact of any fault models for modern memory devices on the SNN fault-tolerance will be investigated and analyzed before applying the curative action/solution (i.e., FAM or FATM). Moreover, the most recent studies that apply reduced-voltage technique for modern memory devices also show that, the fault rate increases as the voltage is reduced [CYG+17a, CYG+17b, SSUCK18, YSE+22], whose trends are similar to the fault models considered in this work. These facts corroborate that our proposed methodology is applicable for any fault models of modern off-chip and on-chip memory devices.

The algorithm complexity of ReSpawn can be derived from its mapping solution. Here, the bit-shuffling operations are performed when storing weights on the faulty memory cells. Such bit-shuffling operations have constant time and space complexity. Therefore, the algorithm complexity of ReSpawn mapping solution can be stated as $O(n)$ for time complexity with $n$ denotes the number of SNN weights, and $O(1)$ for space complexity. Here, optimal solutions that mitigate faults to maintain accuracy without incurring overheads are not feasible, as ReSpawn employs faulty memory cells to allow errors/approximations in computation that lead to a slightly reduced yet acceptable accuracy, while maintaining throughput and incurring minimum overheads (e.g., in energy consumption).

### 5.2.9 Summary of ReSpawn Framework

We propose a novel ReSpawn framework for mitigating the faults in the off-chip and on-chip weight memories for SNN-based systems through SNN fault tolerance analysis, fault-aware mapping, and fault-aware training and mapping. The experimental results show that, ReSpawn with fault-aware mapping improves the accuracy without retraining (e.g., by up to 70% for a 900-neuron network) by minimizing the changes in weight values when storing weight bits in the off-chip and on-chip weight memories. Therefore, our work enhances the SNN fault tolerance with minimum energy overhead, thereby potentially improving the yield of SNN hardware chips.

## 5.3 RescueSNN: Enabling Reliable Executions on SNN Accelerators against Permanent Faults

This section aims at addressing **Problem-2** with the solution for efficiently mitigating permanent faults in the compute engine of SNN neuromorphic accelerators.

### 5.3.1 Motivational Study

To understand the impact of permanent faults in the synapses (local weight registers) of neuromorphic accelerators on the accuracy, we perform experiments that explore different fault rates in the local weight registers while considering the typical architecture of neuromorphic accelerators shown in Figure 2.14. We consider 256x256 synapses with 8-bit precision of weights, and 256 neurons. We assume all neurons are not faulty, and inject permanent faults (i.e., stuck-at 0 or 1) into the weight registers with random distribution and different rates of faulty memory cells, to see the significance of faulty registers on accuracy. For the network, we consider the FC-based SNN architecture in Figure 4.8(a) with 400 excitatory neurons. Further details on the experimental setup are presented in Section 5.3.6. The experimental results are presented in Figure 5.14, from which we make the following key observations.

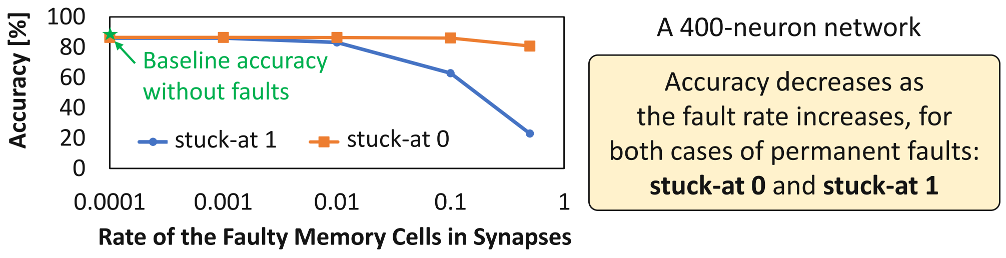


Figure 5.14: The stuck-at faults in the local weight registers can decrease accuracy.

- Classification accuracy decreases as the rate of faulty memory cells increases for both stuck-at 0 and stuck-at 1 scenarios, thereby showing the negative impact of permanent faults in the synapses.
- In the stuck-at 0 case, the stored weight value will either stay the same or decrease from the original value. In the case of decreased weight value, the corresponding neuron will require more stimulus (input spikes) to increase its membrane potential and reach the threshold potential for generating a spike, which represents recognition of a specific class. However, in an SNN model, multiple neurons may be responsible to recognize the same class. Therefore, if the neuron with faulty weight bits cannot recognize the input class, then other neurons may recognize it. As consequence, the accuracy degradation caused by stuck-at 0 in memory cells is relatively small and negligible in some cases.
- In the stuck-at 1 case, the stored weight value will either stay the same or increase from the original value. In the case of increased weight value, the corresponding neuron will require less stimulus (input spikes) to increase its membrane potential and reach the threshold potential for generating a spike, which represents recognition of a specific class. Therefore, this neuron may become more active to generate more spikes for any input classes, which leads to more misclassification. As consequence, the accuracy degradation caused by stuck-at 1 in memory cells is more significant/noticeable than the stuck-at 0 case.
- Combinations of fault types and fault rates lead to different accuracy, which represents different fault patterns in real-world chips, rendering it unpredictable at design time.

### 5.3.2 Scientific Research Challenges

Based on these observations, we outline the following research challenges to devise an efficient solution for the targeted problem.

- *The mitigation technique should not employ retraining*, as retraining needs huge compute and memory costs, processing time, and a training dataset that may not be available in certain cases due to the restriction policies. Moreover, retraining is not a scalable approach considering a large number of fabricated chips, as it needs to consider a unique fault map from each chip thereby retraining per chip. Note, the fault map information can be obtained through the standard wafer/chip test procedure after fabrication, hence this test does not introduce new cost and only incurs a typical cost for chip test [XZLZ20, FCT22].

- *The mitigation should have minimal performance or energy overhead at run time* as compared to that of the baseline design without fault mitigation technique, thereby making it applicable for energy-constrained embedded systems.
- *The technique should not avoid the use of faulty SNN components (i.e., synapses and neurons)*, as it means omitting the entire computations in the respective columns of the SNN compute engine, which leads to throughput reduction.
- *SNNs require a specialized permanent fault mitigation technique* as compared to other neural network computation models (e.g., deep neural networks), since SNNs have different operations and dataflows.

***Required:*** *A low-cost technique for mitigating the negative impact of permanent faults in the compute engine of neuromorphic accelerators, thus enabling reliable execution of SNN inference in an energy-efficient manner.*

### 5.3.3 Novel Contributions

To address the above challenges, we propose *RescueSNN, a novel methodology that enables reliable executions on SNN accelerators under permanent faults.* To the best of our knowledge, this work is the first effort that mitigates permanent faults in the SNN accelerators/chips. Following are the key steps of the RescueSNN methodology (the overview is shown in Figure 5.15).

1. **Analyzing the SNN fault tolerance** to understand the impact of faulty components (i.e., synapses and neurons) on accuracy considering the given fault rates.
2. **Employing the fault-aware mapping (FAM) techniques** to safely map SNN weights and neuron operations to the faulty compute engine, thereby maintaining accuracy and throughput. Our FAM techniques leverage the fault map of the compute engine to perform the following key mechanisms.
   a) Mapping the significant weight bits on the non-faulty memory cells of the synapses (weight registers) to minimally pollute/change the weight values.
   b) Selectively employing faulty neurons that do not cause significant accuracy degradation at inference, based on the behavior of their membrane potential dynamics and spike generation.
3. **Devising simple hardware enhancements** to enable efficient FAM techniques. Our enhancements shuffle the weight bits from the synapses by employing simple combinational logic units (such as multiplexers), so that these weight bits can be used for SNN computations.

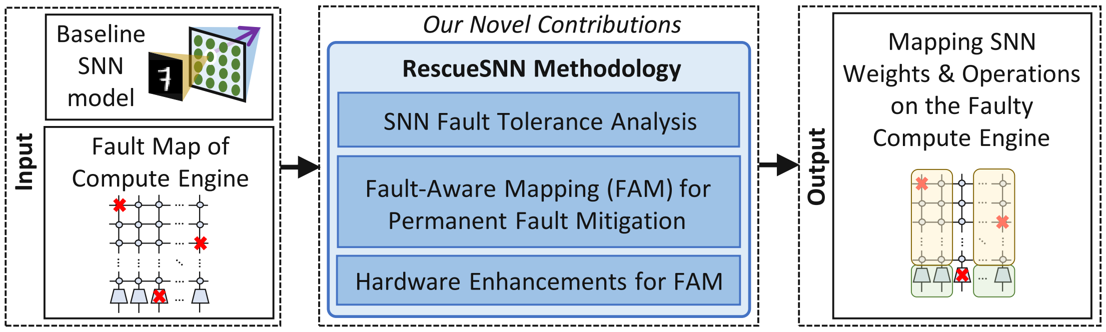

Input
Baseline SNN model
Fault Map of Compute Engine
Our Novel Contributions
RescueSNN Methodology
SNN Fault Tolerance Analysis
Fault-Aware Mapping (FAM) for Permanent Fault Mitigation
Hardware Enhancements for FAM
Output
Mapping SNN Weights & Operations on the Faulty Compute Engine

Figure 5.15: An overview of our novel contributions, which are shown in blue boxes.

### 5.3.4 Permanent Fault Modeling

**Overview:** An SNN compute engine consists of two main components, i.e., synapses and neurons, which have different hardware circuitry. Therefore, we need to define a fault model for each component to achieve fast design space exploration.

1. **Synapses:** Each synapse hardware uses a register to store a weight value. Therefore, each permanent fault in a synapse can affect a single weight bit in the form of either a stuck-at 0 or a stuck-at 1 fault.

2. **Neurons:** Each neuron hardware depends on the neuron model to facilitate its operations. Therefore, permanent faults can manifest in different forms depending on the type of operation being executed on the neuron hardware, as discussed in the following (see an overview in Figure 5.16).

   - *Faults in the '$V_{mem}$ increase' operation* make the neuron unable to increase its membrane potential. As a result, this neuron cannot generate any spikes (i.e., a dormant neuron).
   - *Faults in the '$V_{mem}$ leak' operation* make the neuron unable to decrease its membrane potential. Hence, this neuron acts like the Integrate-and-Fire (IF) neuron model.
   - *Faults in the '$V_{mem}$ reset' operation* make the neuron unable to reset its membrane potential. As a result, this neuron will continuously generate spikes.
   - *Faults in the 'spike generation'* make the neuron unable to generate spikes (i.e., dormant neuron).

**Fault Generation and Distribution:** Previous studies have shown that permanent faults occur in random locations of a chip, thereby leading to a certain fault map [ZGBG18, RFZJ13, WNL16, MKK$^+$20, SSL11]. Following are the key steps to generate and distribute permanent faults on the SNN compute engine; see an overview of the steps in Figure 5.17.

1. We consider a weight memory cell and a neuron operation as the potential fault locations.

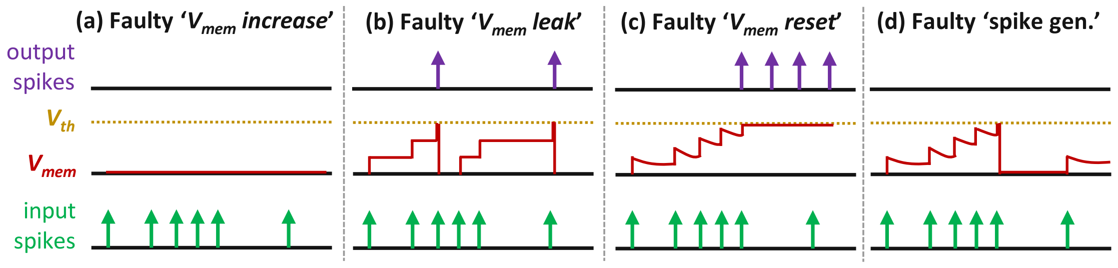

(a) Faulty '$V_{mem}$ increase'
(b) Faulty '$V_{mem}$ leak'
(c) Faulty '$V_{mem}$ reset'
(d) Faulty 'spike gen.'
output spikes
$V_{th}$
$V_{mem}$
input spikes

Figure 5.16: Overview of different faulty LIF neuron operations: (a) faulty '$V_{mem}$ increase', (b) faulty '$V_{mem}$ leak', (c) faulty '$V_{mem}$ reset', and (d) faulty 'spike generation'.

2. We generate permanent faults based on the given fault rate and distribute them randomly across the potential fault locations. The fault rate represents the ratio between the total number of faulty weight memory cells and neuron operations to the total number of potential fault locations (i.e., the total number of weight memory cells and neuron operations).

3. If a fault occurs in a local weight memory cell, then we randomly select the type of stuck-at fault (i.e., either stuck-at 0 or stuck-at 1). Meanwhile, if a fault occurs in a neuron operation, then we randomly select the type of permanent faulty operation.

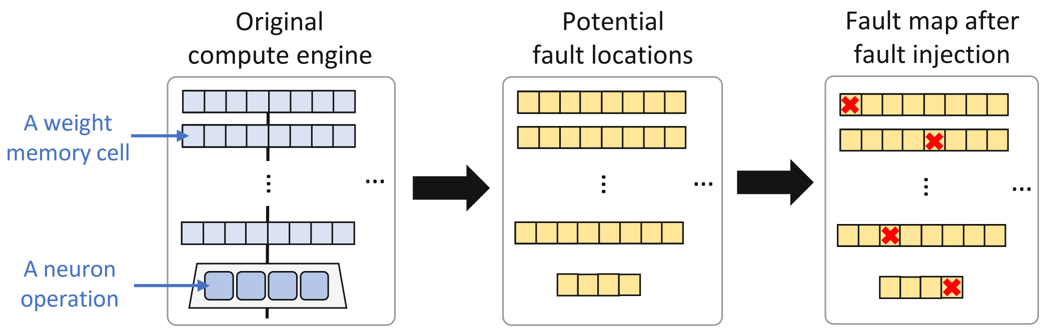


Figure 5.17: The key steps of permanent fault generation and distribution in the SNN compute engine.

### 5.3.5 RescueSNN Methodology

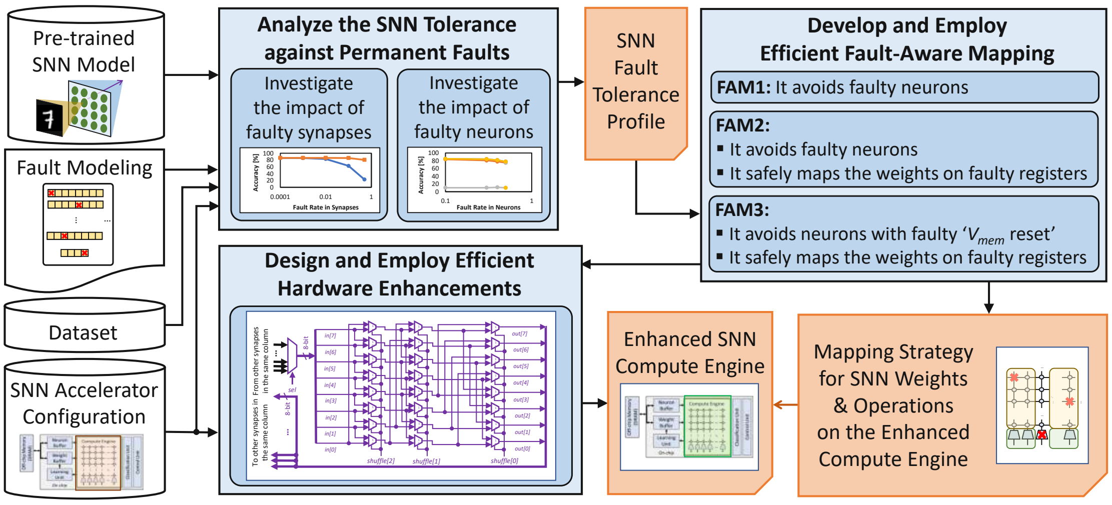
We propose the RescueSNN methodology [PHS23] to mitigate permanent faults in the SNN compute engine with the following key steps; see an overview in Figure 5.18.

1. **SNN fault tolerance analysis (Section 5.3.5.1):** It aims at studying the impact of permanent faults in the compute engine components (i.e., synapses and neurons). Here, we perform experiments that inject the faults into the investigated components with different fault rates. Observations from these experiments are used for devising our fault mitigation technique.

2. **Fault-aware Mapping (FAM) in the SNN compute engine (Section 5.3.5.2):** It leverages the fault map of the compute engine to safely map SNN weights and operations. Our FAM technique employs the following key mechanisms.
   - The significant weight bits are prioritized to be mapped to the non-faulty registers through a bit-shuffling technique, thereby protecting the significant bits from corruption.
   - The faulty neurons can be employed if their behavior does not significantly degrade the final accuracy. These neurons can be selected by observing their membrane potential dynamics and spike generation.

   We leverage these mechanisms to propose three different mapping strategies (i.e., FAM1, FAM2, and FAM3).
   - **FAM1:** It maps weights and operations to the columns of compute engine that do not have faulty neurons.
   - **FAM2:** It is similar to FAM1 strategy, but it also employs a bit-shuffling technique to map the significant weight bits to the non-faulty weight registers.
   - **FAM3:** It shuffles the weight bits to map their significant bits to the non-faulty weight registers, and selectively employs faulty neurons that have tolerable behavior.

3. **Efficient hardware enhancements design (Section 5.3.5.3):** It enables efficient FAM techniques by shuffling the weight bits from the faulty synapses through simple combinational logic units (such as multiplexers), hence these weight bits can be used for SNN computations.



Pre-trained SNN Model
Fault Modeling
Dataset
SNN Accelerator Configuration
Analyze the SNN Tolerance against Permanent Faults
Investigate the impact of faulty synapses
Investigate the impact of faulty neurons
Accuracy [%]
Fault Rate in Synapses
Fault Rate in Neurons
SNN Fault Tolerance Profile
Develop and Employ Efficient Fault-Aware Mapping
FAM1: It avoids faulty neurons
FAM2:
▪ It avoids faulty neurons
▪ It safely maps the weights on faulty registers
FAM3:
▪ It avoids neurons with faulty '$V_{mem}$ reset'
▪ It safely maps the weights on faulty registers
Design and Employ Efficient Hardware Enhancements
To other synapses in the same column
From other synapses in the same column
8-bit
sel
shuffle[2]
shuffle[1]
shuffle[0]
Enhanced SNN Compute Engine
Mapping Strategy for SNN Weights & Operations on the Enhanced Compute Engine

Figure 5.18: An overview of the RescueSNN methodology. The novel contributions are highlighted in blue.

#### 5.3.5.1 SNN Fault Tolerance Analysis under Permanent Fault

SNN fault tolerance analysis is important to understand how a given SNN model will behave considering a specific operating condition (e.g., a combination of certain fault rates, the type of stuck-at fault, the architecture of the compute engine, etc.). This analysis provides information that can be leveraged for devising an efficient fault mitigation technique. Therefore, *our RescueSNN methodology investigates the interaction between the faulty components (i.e., synapses and neurons) and the obtained accuracy.* To do this, we perform the following experimental case studies while considering an FC-based SNN with 400 excitatory neurons, as shown in Figure 4.8(a).

- *We study the accuracy under faulty weight registers*, by injecting a specific stuck-at fault (i.e., either stuck-at 0 or stuck-at 1) into the weight registers, while considering fault-free neurons. Experimental results are shown in Figure 5.14. We observe that both stuck-at 0 and stuck-at 1 faults can degrade accuracy. Therefore, the mitigation technique should address both stuck-at faults.
- *We study the accuracy under faulty neuron operations*, by injecting faults into the neuron hardware to generate faulty neuron operations, while considering fault-free weight registers. Experimental results are shown in Figure 5.19, from which we make the following observations.
  1. Faulty 'spike generation', '$V_{mem}$ increase', and '$V_{mem}$ leak' operations have tolerable accuracy, since their faulty behavior does not dominate the spiking activity, and/or the function of the corresponding faulty neurons for classification may be substituted by other neurons that recognize the same class. Therefore, these neurons can still be used for SNN processing.
  2. Faulty '$V_{mem}$ reset' operations cause significant accuracy degradation, since these operations make the corresponding neurons dominate classification. Therefore, these neurons should not be used for SNN processing.

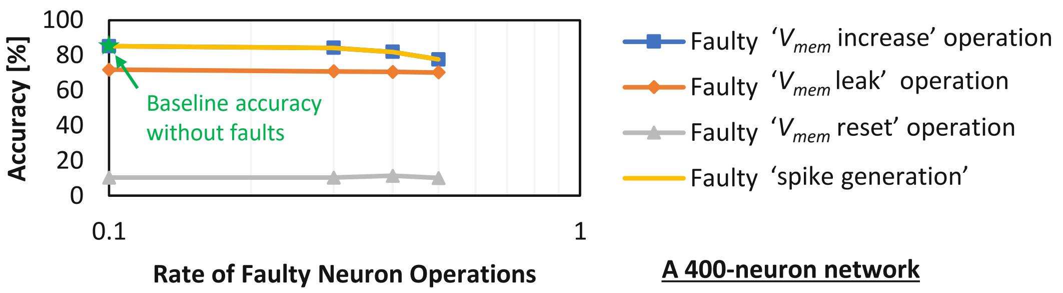


Figure 5.19: Impact of faulty neuron operations on accuracy. Different faulty neuron operations have a different impact on accuracy. Notable accuracy degradation happens when faulty '$V_{mem}$ reset' operations are employed.

Note, complex SNN models with multiple layers and different computational architectures (e.g., convolutional and fully-connected) may have different observation results as compared to results in Figure 5.19. However, previous work has observed similar trends to our study, i.e., neurons with faulty '$V_{mem}$ reset' operations continuously generate spikes (so-called *saturated neurons*) and cause the most significant accuracy degradation than other types of faulty neuron operations [SESA+21]. It also identified that, saturated neurons affect classification accuracy at any layer of SNN models, as these faulty neurons always dominate the classification activity which results in a significant accuracy degradation. This finding is consistent with the insights provided by our study. However, it is still challenging to achieve high accuracy when employing STDP-based learning on complex SNN models [RCK+23], thereby hindering their applicability for diverse applications, such as systems with online training and unsupervised learning requirements (e.g., autonomous mobile agents). Therefore, in this work, we consider the FC-based SNNs shown in Figure 4.8(a) to enable multiple advantages, such as high accuracy, unsupervised learning capabilities, and efficient online training.

#### 5.3.5.2 Our Proposed Fault-Aware Mapping (FAM)

Permanent faults in SNN chips can be identified at the design time and at the run time. The post-fabrication test can be employed to find a set of fault locations (fault map) due to manufacturing defects in the SNN compute engine at the design time [ZGBG18, PHS21a]. Meanwhile, the online test strategy like the Built-In Self-Test (BIST) technique can be employed to obtain the fault map (due to device wear out or physical damages) at the run time [BBD19, WEH+19, MKK+22]. *Our RescueSNN methodology leverages this fault map to safely map the SNN weights and operations to the compute engine, thereby minimizing the negative impact of permanent faults.* To do this, the RescueSNN employs Fault-Aware Mapping (FAM) techniques that mitigate the faults in synapses and neurons through the following key mechanisms.

1. **FAM for Synapses:** The significant weight bits are placed in the non-faulty weight memory cells, while the insignificant bits are placed in the faulty ones, by performing a simple bit-shuffling technique. The significance of weight bits can be identified by experiments that observe the accuracy after modifying a specific bit [PHS21a]. In general, previous studies have observed that the significance of a weight bit is proportional to its bit location. For instance, in 8-bit fixed-point precision, bit-7 has the highest significance than other bits. Furthermore, a synapse may have a single faulty bit or multiple faulty bits. Therefore, we propose a mapping strategy that can address both cases using the following steps (see an overview in Figure 5.20).

   - We identify the faulty weight bits (e.g., through the post-fabrication testing) to obtain information regarding the fault map and fault rate in each synapse hardware.
   - We identify the maximum fault rate in each synapse hardware for safely storing a weight. In this work, we consider a maximum of 2 faulty bits from an 8-bit weight, based on the fault rates that offer tolerable accuracy from analysis in Section 5.3.5.1.
   - We identify the segment in each synapse with the highest number of subsequent non-faulty memory cells. This information is leveraged for maximizing the possibility of

storing the significant bits in the non-faulty cells. Hence, we also examine the corner case (i.e., the right-most and left-most cells) as possible subsequent non-faulty memory cells; see the third row of Figure 5.20(b) with data-3.

- We perform a circular-shift technique for each data word to efficiently implement bit-shuffling.

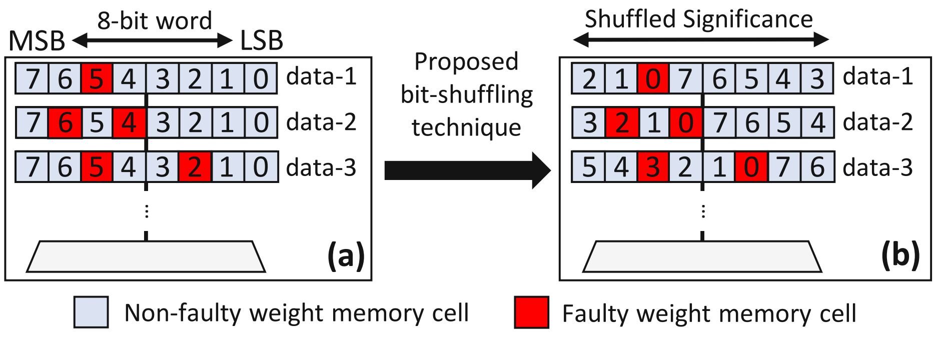


Figure 5.20: (a) Illustration of possible fault locations (fault map) in weight registers. (b) The proposed circular-shift bit-shuffling technique for the corresponding fault map.

2. **FAM for Neurons:** The use of neurons should be avoided if they have faulty '$V_{mem}$ reset' operations, as these faulty operations cause significant accuracy degradation. Meanwhile, neurons with other types of faults can still be used for SNN processing, as their faulty behavior does not dominate the spiking activity. Different SNN operations that aim at recognizing the same input class are mapped to both the faulty and fault-free neurons for maintaining throughput, while compensating the loss from the faulty '$V_{mem}$ increase', '$V_{mem}$ leak', and 'spike generation' operations.

Furthermore, *we leverage these mechanisms for devising three mapping strategies, as the variants of our FAM technique (i.e., FAM1, FAM2, and FAM3)*, which provide trade-offs between accuracy and mapping complexity, as discussed in the following.

- **FAM1:** It avoids mapping the SNN weights and operations to the columns of compute engine that have faulty neurons, as shown in Figure 5.21(a), as faulty neurons can reduce the accuracy more than faulty registers, especially in the case of faulty '$V_{mem}$ reset'. However, FAM1 does not mitigate the negative impact of faults in the registers, hence the accuracy improvement is sub-optimal. The benefit of FAM1 is due to its simple mechanism which enables a low-complexity control mechanism.
- **FAM2:** It maps the SNN weights and operations to the columns of compute engine that have fault-free neurons (just like FAM1) and employs a bit-shuffling technique to map the significant weight bits to the non-faulty memory cells, as shown in Figure 5.21(b). Therefore, FAM2 can improve the SNN fault tolerance at the cost of a more complex control mechanism than FAM1.
- **FAM3:** It selectively maps the SNN weights and operations to the columns of compute engine that do not have faulty '$V_{mem}$ reset' operations, as well as maps the significant

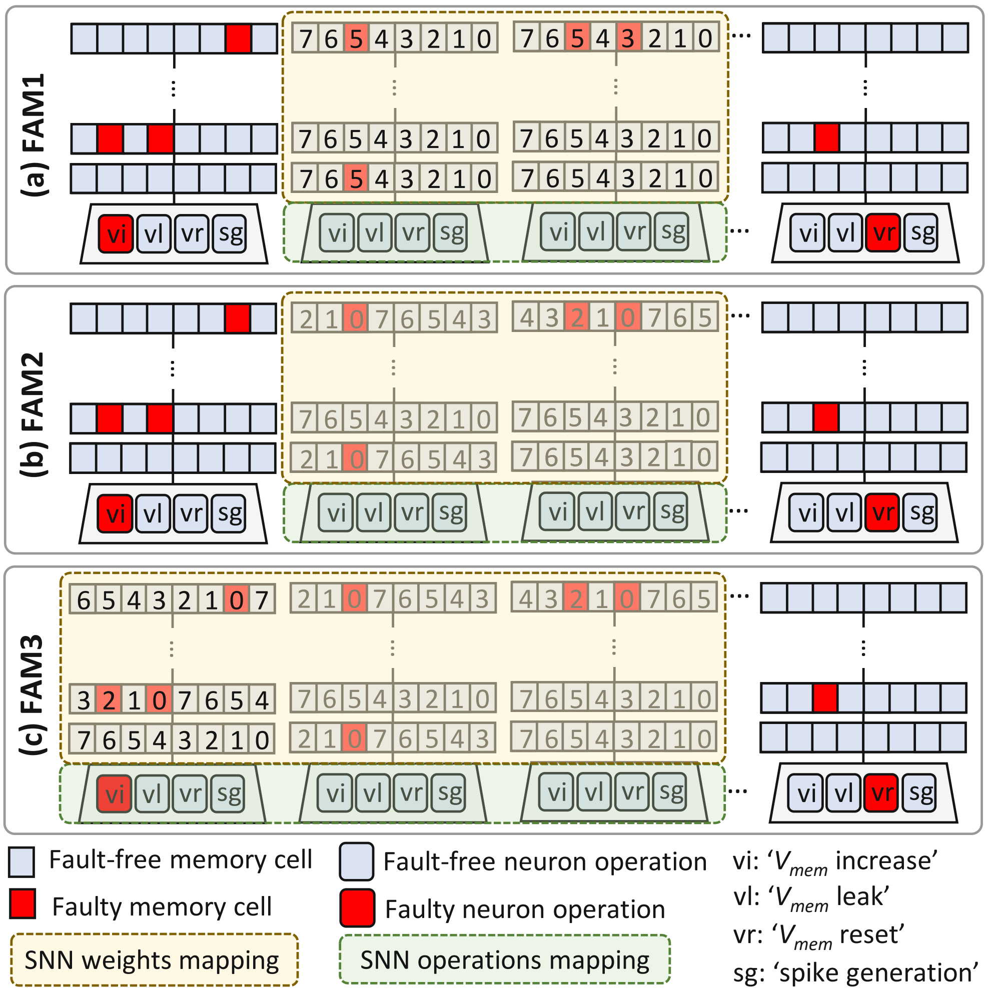


Figure 5.21: Our FAM strategies: (a) FAM1, (b) FAM2, and (c) FAM3.

weight bits to the non-faulty memory cells using a bit-shuffling technique, as shown in Figure 5.21(c). Therefore, FAM3 can enhance the SNN fault tolerance as compared to FAM1 at the cost of a more complex control mechanism, and can improve the throughput as compared to FAM1 and FAM2.

Information regarding how to map the SNN weights and operations to the compute engine is provided through software program (e.g., firmware), thereby enabling the applicability and flexibility of the proposed FAM technique (e.g., FAM1, FAM2, or FAM3) for different possible fault maps in the compute engine. The metadata of this information is stored in the on-chip buffer, which can be accessed for operations in the compute engine.

### 5.3.5.3 Our Hardware Enhancements for FAM

Our FAM2 and FAM3 strategies may make the weight bits stored in a shuffled form. Therefore, an additional mechanism is required for converting these weight bits into the

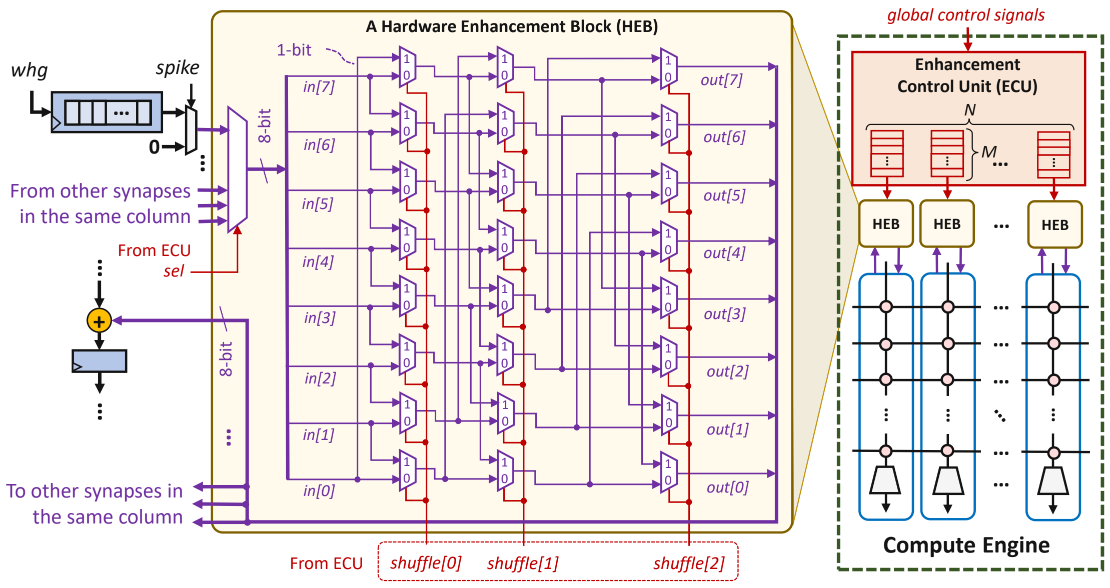


Figure 5.22: The architecture of the proposed enhancements, including the hardware enhancement blocks (HEBs) and the enhancement control unit (ECU), for accommodating FAM strategies.

original order, so that they can be used for SNN executions. Toward this, *we propose lightweight hardware enhancements to support the re-shuffling mechanism to undo the data transformation, i.e., through a simple 8-bit barrel shifter*. The key idea is to re-shuffle the order of output wires from each synapse into the original order, so that the corresponding weights can be used directly for neuron operations. To optimize the overheads (e.g., area), we share the *hardware enhancement block* (HEB) with all synapses in the same column of compute engine, and different synapses will access the enhancement block at different times; see Figure 5.22. In this manner, the number of HEBs is equal to the number of columns in the SNN compute engine. Furthermore, to control the functionality of HEBs, we employ an *enhancement control unit* (ECU). This ECU stores the bit-shifting information and uses it for controlling the barrel shifter in HEBs. For each column of the compute engine, the ECU employs (1) a dedicated selector signal *sel* to determine which weight should be processed in the HEB at a time, and (2) a set of registers that stores bit-shifting information *shuffle[2:0]* for all weights in the same column.

### 5.3.6 Evaluation Methodology

For evaluating the RescueSNN methodology, we employ the experimental setup shown in Figure 5.23. We use the FC-based network shown in Figure 4.8(a) with a different number of neurons, to evaluate the generality of our RescueSNN methodology. For conciseness, we represent a network with $i$-number of neurons as N$i$. We use the MNIST and Fashion MNIST datasets as the workloads, and adopt the same test conditions as used widely by the SNN community [DC15]. For comparison, we consider the SNN without fault

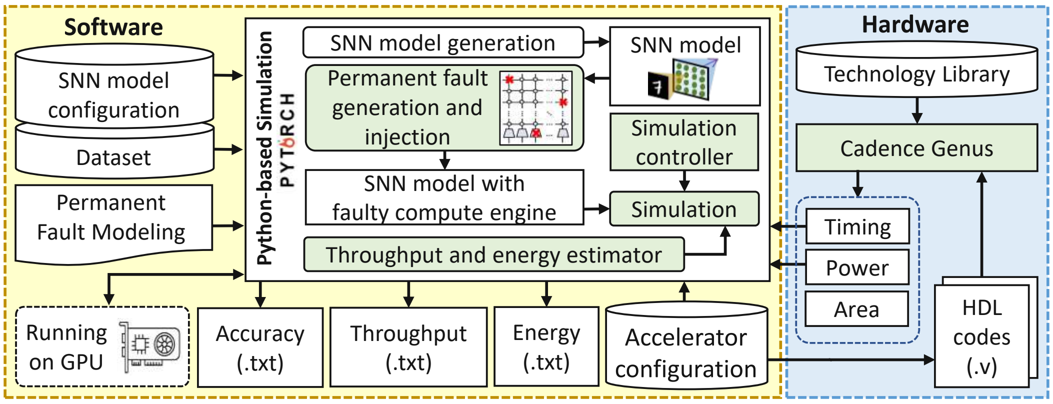


Figure 5.23: Overview of the experimental setup and tools flow.

mitigation as the baseline.

**Fault Generation and Injection:** Permanent faults are generated based on the fault modeling in Section 5.3.4. To do this, we first generate binary values (i.e., 0 and 1) based on the given fault rate while considering the potential fault locations (shown in Figure 5.17). Here, '0' represents a non-faulty memory cell in synapses or a non-faulty operation in neurons; while '1' represents a faulty memory cell in synapses or a faulty operation in neurons. These binary values are then randomly distributed into an array that represents the potential fault locations, so that each value corresponds to a specific weight memory cell or a specific neuron operation. Figure 5.24 shows the potential locations/components that can be affected by permanent faults to cause faulty memory cells as well as faulty '$V_{mem}$ increase', '$V_{mem}$ leak', '$V_{mem}$ reset', and 'spike generation' operations. For *each fault in the weight memory cells (synapses)*, we randomly determine the type of fault (either stuck-at 0 or stuck-at 1). In stuck-at 0 case, value 0 is injected into the corresponding memory cell; while in stuck-at 1 case, value 1 is injected. Meanwhile, *each fault in neurons* corresponds to either faulty '$V_{mem}$ increase', '$V_{mem}$ leak', '$V_{mem}$ reset', or 'spike generation' operation. Each faulty behavior in the corresponding neuron is realized through different approaches as described in the following.

- *Faulty '$V_{mem}$ increase' operation*: It is mainly caused by faulty addition in the '$V_{mem}$ increase' part, hence $V_{mem}$ is not increased despite there are incoming spikes.
- *Faulty '$V_{mem}$ leak' operation*: It is mainly caused by faulty subtraction in the '$V_{mem}$ leak' part, hence $V_{mem}$ is not decreased despite there are no incoming spikes.
- *Faulty '$V_{mem}$ reset' operation*: It is mainly caused by faulty comparison in the '$V_{mem}$ reset' part, hence the spike generator is activated to continuously generate spikes.
- *Faulty 'spike generation'*: It is mainly caused by faulty multiplexing in the 'spike generation' part, hence the spike generator is always deactivated and no output spikes are produced.

**Accuracy Evaluation:** We use the Python-based simulations [HSK+18], which run on Nvidia RTX 2080 Ti GPUs, while considering the SNN accelerator shown in Figure 2.14.

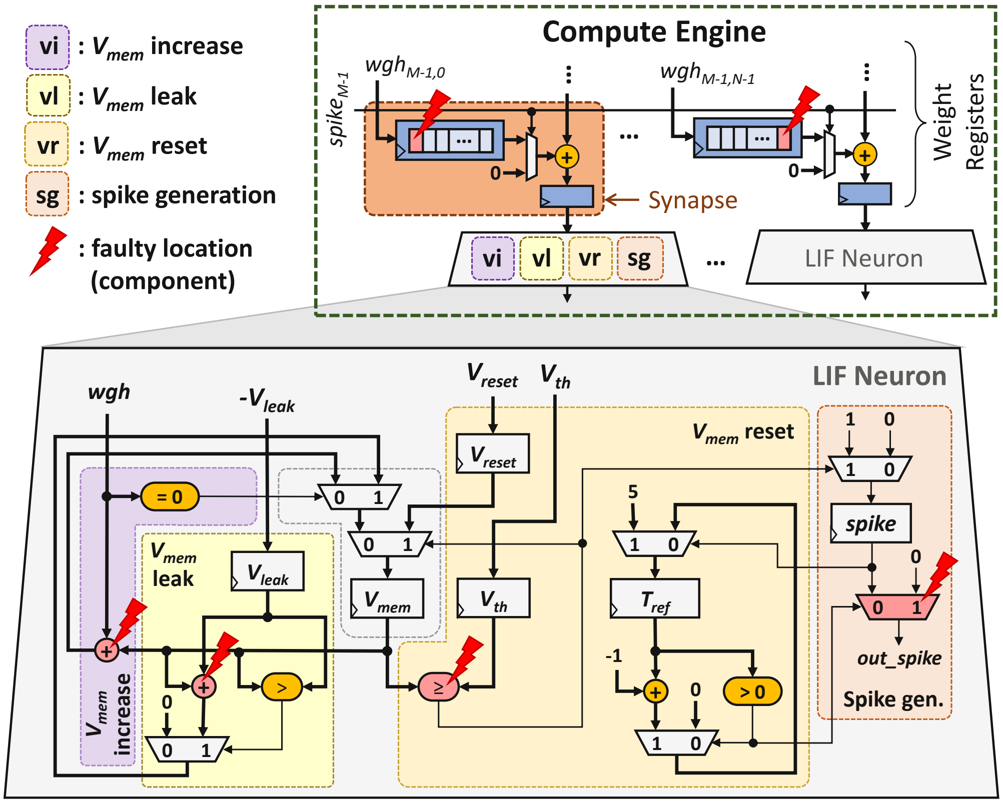


Figure 5.24: The potential locations/components that can be affected by permanent faults to cause faulty memory cells as well as faulty '$V_{mem}$ increase', '$V_{mem}$ leak', '$V_{mem}$ reset', and 'spike generation' operations.

**Hardware Evaluation:** We evaluate the area, energy consumption, and throughput of both the original compute engine (without enhancements) and the enhanced compute engine using our RescueSNN methodology. To do this, we design RTL codes for both the original and enhanced compute engines, then synthesize them using the Cadence Genus tool considering a 65 nm CMOS technology to obtain their area, power consumption, and timing (i.e., a clock cycle latency for SNN processing on the compute engine). Afterward, we calculate the required number of cycles and computation latency for processing an input sample (i.e., *latency-per-sample*), considering the timing from synthesis and the mapping strategy on active synapses and neurons in the compute engine. Then, we estimate the throughput by computing the number of samples that can be processed within one second of SNN inference based on the information of latency-per-sample. Furthermore, we also estimate the energy consumption by leveraging the information regarding the power consumption from synthesis and the latency-per-sample for SNN inference. The estimation of throughput and energy consumption is also performed on top of the Python-based simulation framework.

### 5.3.7 Results and Discussion

#### 5.3.7.1 Maintaining the Accuracy

Figure 5.25 presents the experimental results for the accuracy of different fault mitigation techniques, i.e., the baseline and our FAM-based strategies including FAM1, FAM2, and FAM3. We observe that the baseline suffers from a significant accuracy degradation as shown by ❶, because it does not mitigate faults in synapses and neurons, thereby leading to unreliable SNN executions. The significant accuracy degradation is mainly due to the fault model for faulty '$V_{mem}$ reset' operation that makes the corresponding neuron generate spikes continuously once its membrane potential $V_{mem}$ reaches the threshold potential $V_{th}$, thereby dominating the classification activity and leading to high misclassification. We also observe that FAM1 significantly improves the SNN fault tolerance as compared to the baseline, because FAM1 avoids the use of faulty neurons, especially for faulty '$V_{mem}$ reset' operations, as shown by ❷. Our FAM2 improves the SNN fault tolerance even more as compared to FAM1, since FAM2 also mitigates faults in the weight registers in addition to avoiding the use of faulty neurons, as shown by ❸. Meanwhile, our FAM3 also significantly improves the SNN fault tolerance from baseline and FAM1, and obtains comparable accuracy to FAM2, since FAM3 mitigates faults in weight registers and selectively uses faulty neurons. It achieves up to 80% accuracy improvement as compared to the baseline for the MNIST dataset, as shown by ❹. We also observe that the same reasons are also applicable to different workloads, thereby leading the accuracy profiles for the Fashion MNIST to have similar trends to the accuracy profiles for the MNIST. These results show that *our FAM strategies (FAM1, FAM2, and FAM3) are effective for mitigating permanent faults in the SNN compute engine without retraining, across different model sizes, fault rates, and workloads.*

#### 5.3.7.2 Maintaining the Throughput

Figure 5.26 presents the experimental results for the throughput of different mitigation techniques, i.e., the baseline and our FAM strategies (FAM1, FAM2, and FAM3). We observe that the baseline has the highest throughput across different model sizes and fault rates, as it uses all synapses and neurons for performing SNN executions, as shown by ①. Meanwhile, FAM1 and FAM2 may suffer from throughput reduction because they avoid the use of faulty neurons, thereby omitting the corresponding columns of the SNN compute engine. For instance, FAM1 and FAM2 may suffer from 30% throughput reduction for N1600 with 0.1 fault rate, as shown by ②. Meanwhile, our FAM3 can maintain the throughput close to the baseline (e.g., keeping the throughput reduction below 25% in a 0.5 fault rate), thereby improving the throughput significantly as compared to FAM1 and FAM2. The reason is that, FAM3 omits the columns of compute engine only if the corresponding neurons have faulty '$V_{mem}$ reset' operations. For instance, FAM3 has less than 15% throughput reduction for N1600, as indicated by ③. These results show that *our FAM3 is effective for maintaining the throughput of SNN compute engine with permanent faults across different model sizes, fault rates, and workloads.*

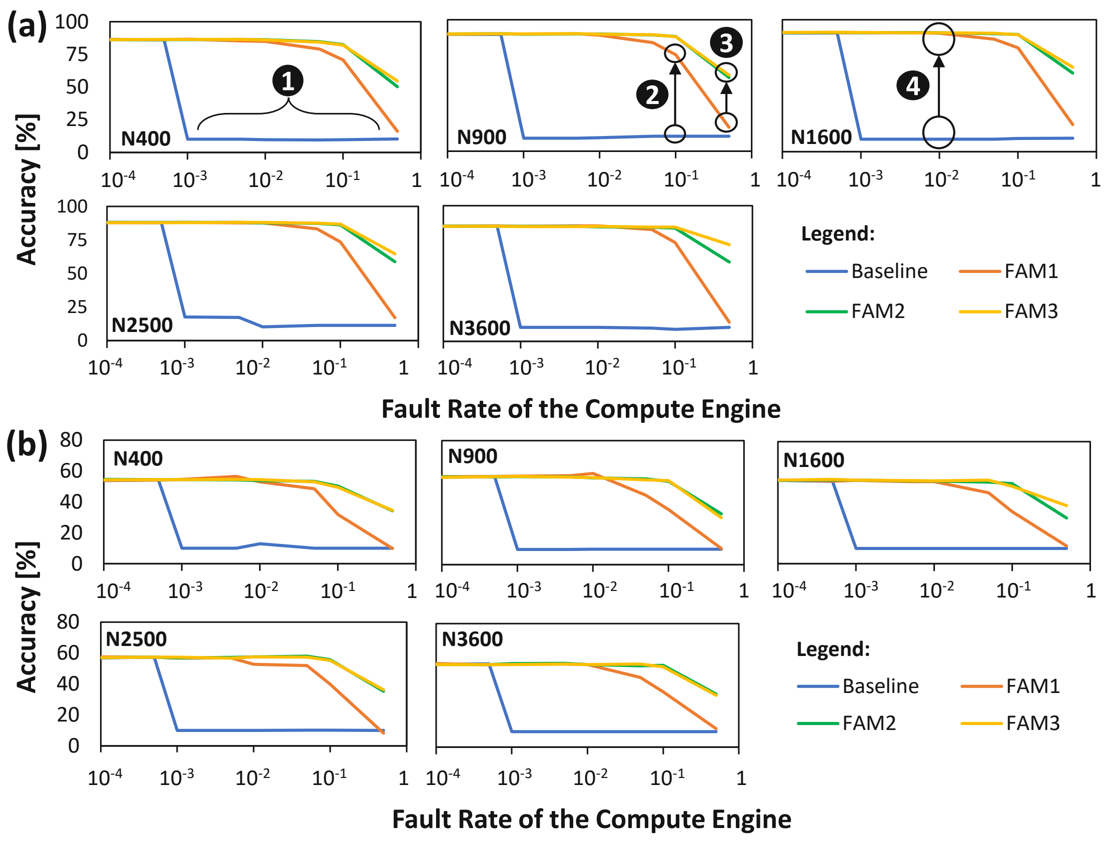


Figure 5.25: Accuracy profiles for different mitigation techniques (i.e., baseline, FAM1, FAM2, and FAM3), different model sizes (i.e., N400, N900, N1600, N2500, and N3600), different fault rates, and different workloads: (a) MNIST and (b) Fashion MNIST.

#### 5.3.7.3 Energy Consumption and Area Overheads

Figure 5.27 shows the experimental results for the energy consumption of different mitigation techniques, i.e., the baseline and our FAM strategies (FAM1, FAM2, and FAM3). We observe that different techniques have comparable energy for small fault rates, as shown by label-④. The reason is that small fault rates have a low probability of faulty neurons, hence the resource utilization for different techniques is similar. For large fault rates, FAM1 and FAM2 have higher energy consumption than the baseline and FAM3, as shown by label-⑤. The reason is that large fault rates have a high probability of faulty neurons, hence the resource utilization for different techniques is different, i.e., FAM1 and FAM2 avoid the use of faulty neurons, thereby incurring higher compute latency and energy consumption. The baseline and FAM3 have comparable energy since FAM3 employs simple hardware enhancements: (1) multiplexing operations in each HEB which are shared for all synapses in the same column of the compute engine, and (2) registers accesses in ECU, thereby minimizing the energy consumption overhead for FAM3 (i.e., within 30%). For area footprint, the original compute engine consumes around

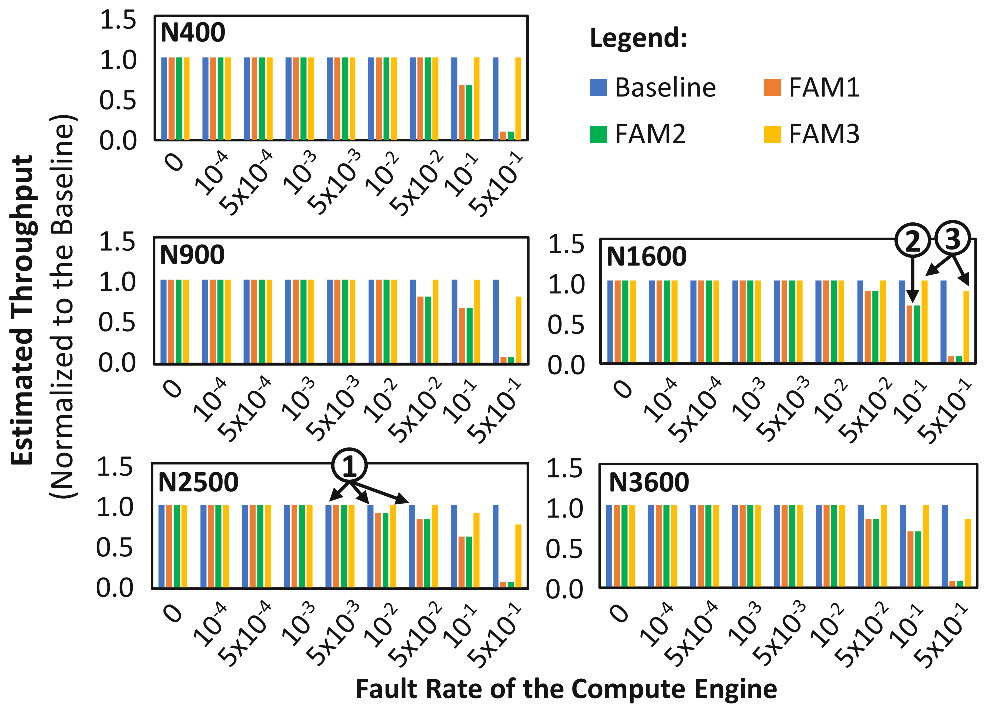


Figure 5.26: Throughput across different mitigation techniques, different model sizes, and different fault rates for both the MNIST and Fashion MNIST datasets, as they have a similar number of SNN weights and operations.

6.27 $mm^2$ of area, while the one with proposed enhancements consumes around 8.56 $mm^2$ of area. Hence, the proposed enhancements incur about 36.5% of area overhead, which encompasses about 36.2% of ECU and about 0.3% of HEBs. The area of ECU dominates the total area of enhancements since it mainly employs a set of 3-bit registers (i.e., 256x256 registers), which incurs a larger area as compared to HEBs (i.e., 256x25 multiplexers). These results show that *our FAM3 achieves minimum overheads in terms of energy consumption and area for the SNN compute engine across different model sizes, fault rates, and workloads.*

In summary, the above discussions show that *our RescueSNN methodology can effectively mitigate permanent faults in the SNN chips without retraining.* Since our RescueSNN addresses permanent faults during both the design time and the run time, it increases the yield of SNN chips, as well as enables efficient and reliable SNN executions during their operational lifetime. Furthermore, our RescueSNN also avoids carbon emission as it does not need any retraining, thus offering an environment-friendly solution [SGM19, SGM20].

#### 5.3.7.4 Further Discussion

In general, we observe that a faulty '$V_{mem}$ reset' operation can cause significant accuracy degradation as it deteriorates the neuron from the expected behavior. The reason is that,

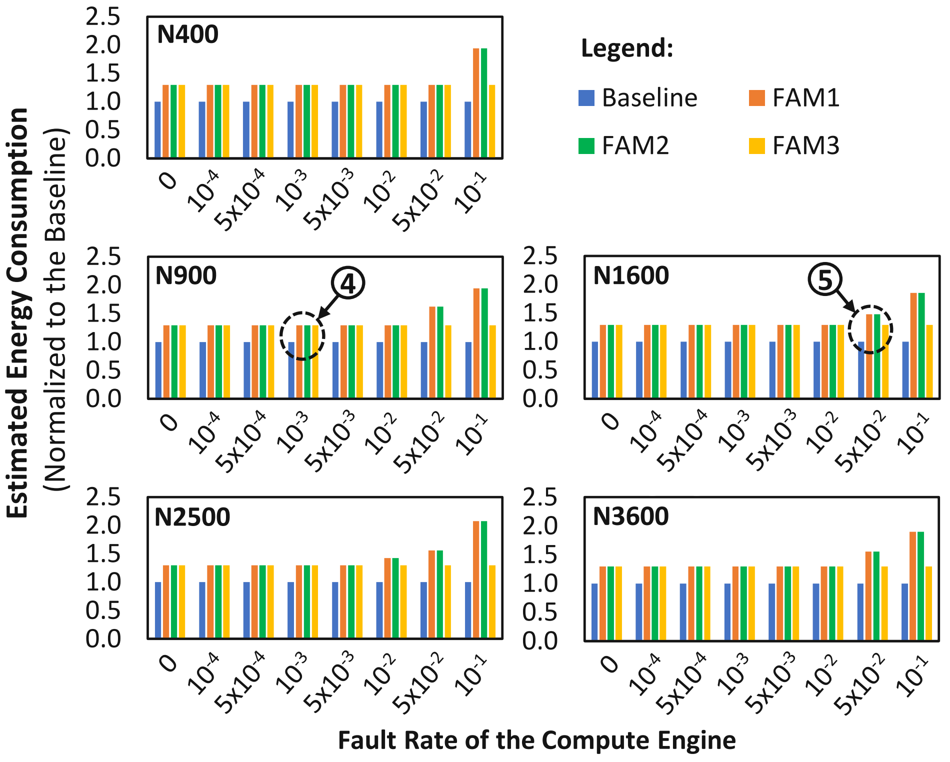


Figure 5.27: Energy consumption across different mitigation techniques, different model sizes, and different fault rates for both MNIST and Fashion MNIST, as they have a similar number of SNN weights and operations.

the generated (faulty) spikes will affect how the SNN model understands the information, since an SNN model typically employs a certain spike coding scheme, i.e., rate coding in this work. Therefore, a neuron with faulty '$V_{mem}$ reset' operation will generate a high number of spikes and dominate the classification activity, thereby leading to high misclassification and significant accuracy degradation. We also observe that, a higher number of spikes generated by faulty '$V_{mem}$ reset' operation also indicates that the SNN model performs more frequent neuron operations that correspond to spike generation. This condition leads to higher power/energy consumption for SNN processing, which has been observed and studied in previous works [KSVR19, PKNY20, PS23b].

**Comparison with Retraining Technique:** In a standard chip fabrication process, manufactured chips are evaluated in a wafer/chip test procedure (i.e., wafer acceptance test and chip probing test). This test procedure aims at evaluating the quality of each chip, including any faults in the chip [XZLZ20, FCT22]. In this step, the permanent faults and the corresponding fault map information from manufacturing defects are identified. Therefore, this step does not introduce new cost, and only requires a typical cost for a standard wafer/chip test procedure [XZLZ20, FCT22]. In the retraining technique, the fault map information is then incorporated in the retraining process considering how the weights and neuron operations are mapped on the SNN compute engine, i.e., so-called *fault-aware training* (FAT). In this manner, the SNN model is expected to adapt to the presence of faults, hence maintaining high accuracy. This indicates that, the retraining technique requires (1) fault map information from the chip test procedure, and (2) a full training dataset, which may be unavailable due to restriction policy. *Furthermore, each chip has a unique fault map which requires its own retraining process, thereby incurring huge time and energy costs.* Otherwise, the retraining technique will not be effective. Meanwhile, our proposed FAM technique in RescueSNN methodology leverages the fault map information to safely map the weights and neuron operations on the SNN compute engine. It ensures that the SNN processing is not negatively affected by permanent faults, thereby maintaining high accuracy. Although each chip has a unique fault map which requires a specific mapping, the cost of devising the mapping strategy is significantly lower than the cost of retraining. Furthermore, our FAM technique does not require any training dataset, hence it is highly applicable to a wide range of SNN applications.

**Benefits and Limitations of Pruning:** Neurons in the FC-based SNN architecture shown in Figure 4.8(a) can be pruned while keeping the accuracy close to that of the original network, considering that a high rate of faulty '$V_{mem}$ increase' operations does not significantly degrade accuracy. The benefits of pruning in FC-based architecture have been demonstrated in previous work [RPR19], including reduction of memory footprint and energy consumption. The pruning technique is suitable if we rely on offline training, i.e., an SNN model is trained offline with the training dataset, and the knowledge learned from the training phase is kept unchanged during inference at run time. However, the pruning technique is not suitable if we consider SNN-based systems that need to update their knowledge regularly at run time to adapt to different operational environments (i.e., dynamic environments) such as autonomous mobile agents (e.g., UGVs). The reason is that, SNN-based systems may encounter new input features in different environments and the offline-trained knowledge may not be representative for recognizing the corresponding classes, thereby leading to low accuracy at run time and requiring online training to update their knowledge [PS21b, PS22a]. Therefore, SNN models with unpruned neurons and unsupervised learning capabilities are beneficial for learning and recognizing new features in (unlabeled) data samples from the operational environments during online training. In summary, the users can select which SNN model to employ depending on the design requirements. An alternative is employing the FC-based SNNs shown in Figure 4.8(a) with/without pruning since they can enable multiple benefits, such as high accuracy when employing STDP-based learning under unsupervised settings, and efficient online training capabilities.

**Algorithm Complexity:** The algorithm complexity of RescueSNN can be derived from its mapping solution. Here, the bit-shuffling operations are performed when storing weights on the faulty synaptic registers, and the selective mapping operations are performed when storing neuron parameters on the faulty neurons. Such bit-shuffling and selective mapping operations have constant time and space complexity. Therefore, the algorithm complexity of RescueSNN mapping solution can be stated as $O(n)$ for time complexity with $n$ denotes the number of SNN parameters (i.e., weight and neuron parameters), and $O(1)$ for space complexity. Here, optimal solutions that mitigate faults to maintain accuracy without incurring overheads are not feasible, as RescueSNN employs faulty memory cells and faulty neurons to allow errors/approximations in computation that lead to a slightly reduced yet acceptable accuracy, while maintaining throughput and incurring minimum overheads (e.g., in area and energy consumption).

### 5.3.8 Summary of RescueSNN Methodology

We propose a novel RescueSNN methodology for mitigating permanent faults in the compute engine of SNN accelerators/chips. Our RescueSNN leverages the fault map of the compute engine to perform fault-aware mapping for SNN weights and operations, and employs efficient hardware enhancements for the proposed mapping technique. The results show that RescueSNN effectively improves the SNN fault tolerance against permanent faults without retraining. As a result, faulty SNN chips can be rescued and used for reliable SNN processing during their operational lifetime.

## 5.4 SoftSNN: Low-Cost Fault Tolerance for SNN Accelerators against Soft Errors

This section aims at addressing **Problem-3** with the solution for efficiently mitigating soft errors in the compute engine of SNN neuromorphic accelerators.

### 5.4.1 Motivational Study

To understand the impact of soft errors in the compute engine of neuromorphic accelerators on the accuracy, we perform experiments with the typical architecture of neuromorphic accelerators shown in Figure 2.14. We first perform fault injection into the local weight registers (synapses) of different neurons with random distribution, while considering different fault maps and different fault rates. Here, we consider 256x256 synapses with 8-bit precision of weights, and 256 neurons. For the network, we consider the FC-based SNN architecture in Figure 4.8(a) with 400 excitatory neurons. Further details on the experimental setup are presented in Section 5.4.6. The experimental results are presented in Figure 5.28, from which we make the following key observations.

- Different combinations of fault maps and fault rates (which represent different possible soft error patterns in real-world conditions) lead to diverse accuracy profiles, even for the same SNN model and workload, indicating its unpredictable nature at design time; see Ⓐ in Figure 5.28(a).
- A potential solution is to employ redundant executions (i.e., re-execution) as it does not require hardware modification, but at the cost of huge latency and energy overheads; see Figure 5.28(b).

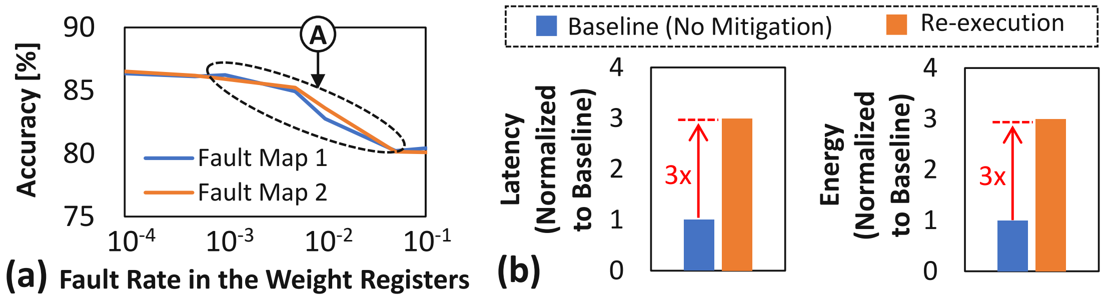


Figure 5.28: Results of a 400-neuron network on the MNIST for (a) accuracy, considering different fault locations (fault maps) and fault rates in the weight registers of the compute engine, and (b) latency and energy for different designs.

## 5.4.2 Scientific Research Challenges

Based on these observations, we highlight the following *research challenges* in devising solutions for the targeted problem.

- *The mitigation technique should recognize any faulty components (weight registers and neurons operations) at run time*, to cope with unpredictable run-time scenarios of different soft error profiles.
- *The mitigation should not employ re-execution*, because it requires huge latency and energy overheads.
- *The mitigation should have minimal latency and energy overheads* compared to that of the "SNN without mitigation", thereby making it applicable for latency- and energy-constrained applications.

***Required:*** *A lightweight mitigation technique for soft errors in the compute engine of neuromorphic accelerators, thereby enabling reliable execution of SNN inference in an energy-efficient manner.*

## 5.4.3 Novel Contributions

To address the above challenges, we propose *SoftSNN, a novel methodology that enables reliable SNN processing on hardware accelerators under soft errors without re-execution.* To the best of our knowledge, this work is the first effort that studies the impact of soft errors in the SNN accelerators, and develops a cost-effective mitigation technique. Our SoftSNN employs the following key steps; see an overview in Figure 5.29.

1. **Analyzing the SNN fault tolerance under soft errors** to understand the impact of faulty SNN components (i.e., synapses and neurons) on the accuracy under different fault rates.
2. **Employing different Bound-and-Protect (BnP) techniques** to bound the weight values within a safe range, and protect the neurons from performing faulty operations, based on the information from the SNN fault tolerance analysis.

3. **Employing lightweight hardware support for the BnP techniques** to efficiently identify when faulty weight values and neuron operations occur, then perform weight bounding and generate safe neuron behavior that does not cause significant accuracy degradation.

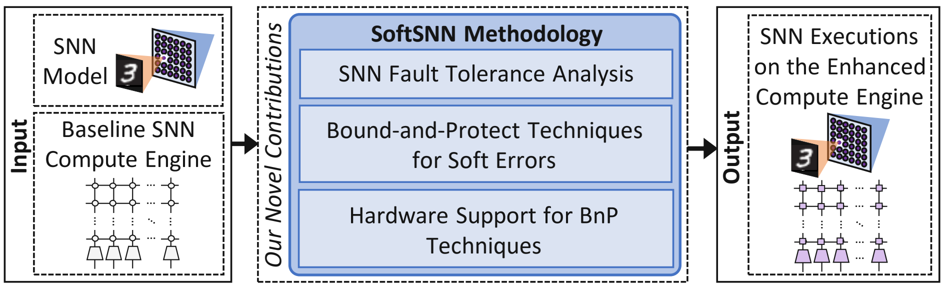

Input
SNN Model
Baseline SNN Compute Engine
Our Novel Contributions
SoftSNN Methodology
SNN Fault Tolerance Analysis
Bound-and-Protect Techniques for Soft Errors
Hardware Support for BnP Techniques
Output
SNN Executions on the Enhanced Compute Engine

Figure 5.29: Overview of our novel contributions, which are highlighted in blue.

### 5.4.4 Transiet Fault Modeling

**Overview:** The SNN compute engine consists of synapse and neuron parts, each having specialized hardware circuitry. Therefore, we need to define the transient fault modeling for each part.

1. **Synapse Part:** A fault in a synapse hardware only affects a single weight bit in the form of a bit flip. This faulty bit persists until it is overwritten with a new bit value.
2. **Neuron Part:** Soft errors in the neuron hardware can manifest in different forms depending upon the type of operation being executed on the neuron hardware, as discussed in the following (see an overview in Figure 5.30).
   - *Soft errors in the '$V_{mem}$ increase' operation* make the neuron unable to increase $V_{mem}$, hence this neuron is unable to reach $V_{th}$ and does not produce any spikes.
   - *Soft errors in the '$V_{mem}$ leak' operation* make the neuron unable to decrease $V_{mem}$.
   - *Soft errors in the '$V_{mem}$ reset' operation* make the neuron unable to reset $V_{mem}$, hence this neuron continuously produces spikes.
   - *Soft errors in the 'spike generation'* make the neuron unable to produce any spikes.
   - These faulty operations persist until the neuron parameters are replaced with a new set of parameters.

**Soft Error Generation and Distribution:** Soft errors typically occur in random locations of a chip [Bau05], leading to a certain fault map. Following are the steps for generating and distributing soft errors (see an overview in Figure 5.31).

- We consider each weight memory cell and neuron operation as the potential fault locations.
- We generate soft errors considering the given fault rate, and distribute them randomly across the potential fault locations.

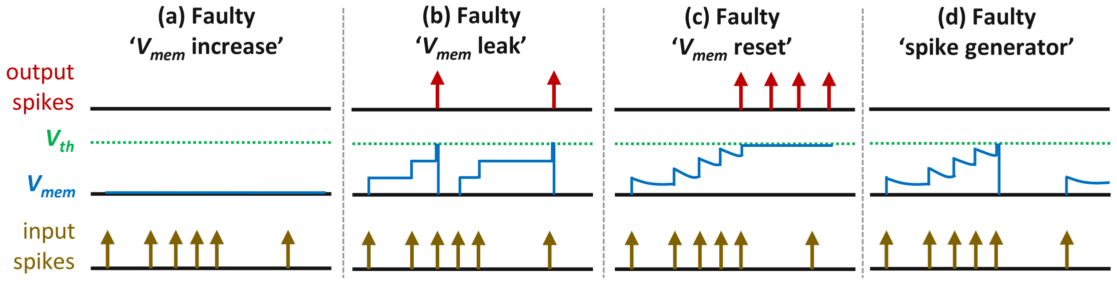


Figure 5.30: Overview of different faulty LIF neuron operations: (a) faulty '$V_{mem}$ increase', (b) faulty '$V_{mem}$ leak', (c) faulty '$V_{mem}$ reset', and (d) faulty 'spike generation'.

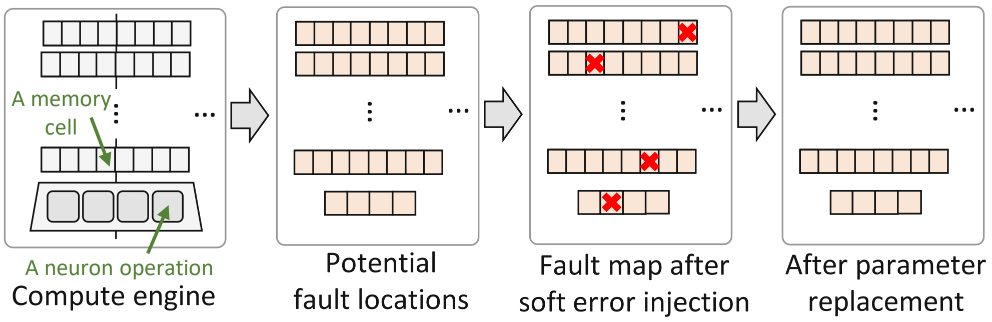

A memory cell
A neuron operation
Compute engine
Potential fault locations
Fault map after soft error injection
After parameter replacement

Figure 5.31: Our steps for conducting soft error generation and distribution in the SNN compute engine.

- If a fault occurs in a memory cell, we flip the stored bit, which persists until it is overwritten by a new value. If an error occurs in a neuron operation, we randomly select the type of faulty operation, which persists until new parameters are set for the respective neuron.

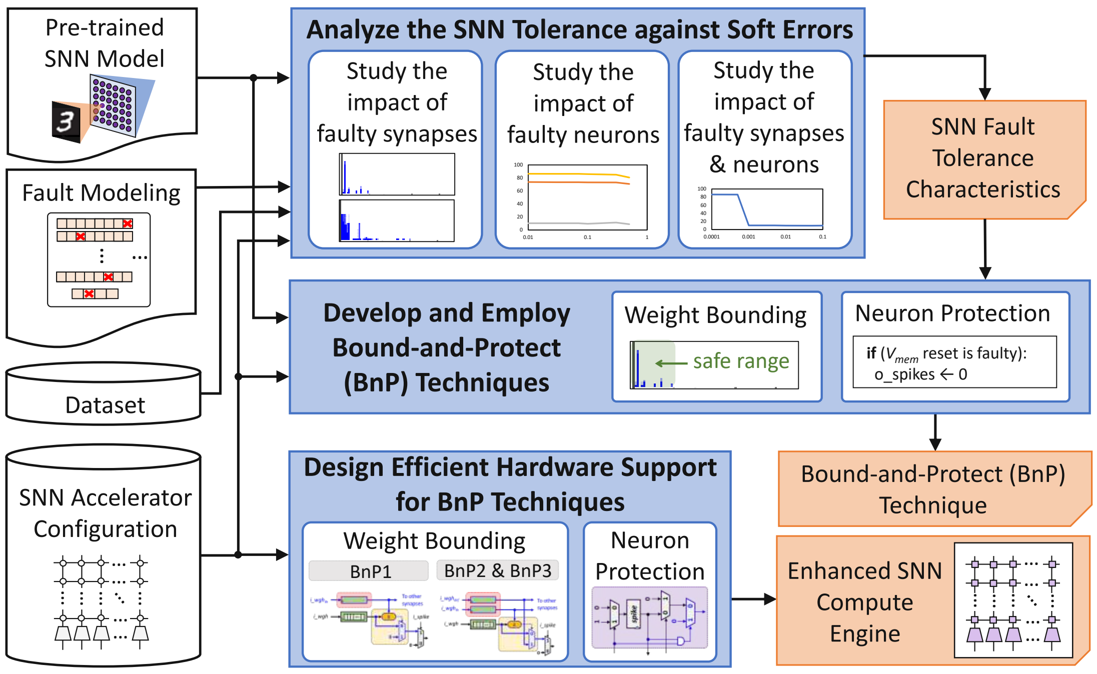
### 5.4.5 SoftSNN Methodology

We propose the SoftSNN methodology [PHS22b] to mitigate soft errors in the SNN compute engine with the following key steps; see an overview in Figure 5.32. A description of each step is then provided in the subsequent sections.

1. **Analysis the SNN fault tolerance under soft errors (Section 5.4.5.1).** It aims at understanding the impact of faulty synapses and/or faulty neurons on the accuracy across different fault rates. Here, we perform experiments that inject the faults into the investigated components with different fault rates. Observations from these experiments are used for devising our fault mitigation technique.
2. **Employment of different BnP techniques (Section 5.4.5.2).** The key ideas of BnP techniques are bounding the weight values within a safe range, and protecting the neurons from performing faulty operations, based on the information from the SNN fault tolerance analysis. We leverage these mechanisms to propose three different strategies (i.e., BnP1, BnP2, and BnP3).

- **BnP1:** It replaces the weights that are greater or equal to the threshold with zero.
- **BnP2:** It replaces the weights that are greater or equal to the threshold with the maximum weight value from clean SNN.
- **BnP3:** It replaces the weights that are greater or equal to the threshold with a highly probable value from the weight distribution of clean SNN.
- **For All BnP Techniques:** We continuously monitor the neuronal dynamics, and if the faulty '$V_{mem}$ reset' operation occurs, we disable the respective spike generation.

3. **Employment of efficient HW enhancements for the BnP techniques (Section 5.4.5.3).** It aims at identifying when faulty weight values and neuron operations happen. If so, it will perform weight bounding and generate safe neuron behavior that does not cause significant accuracy degradation.



Pre-trained SNN Model
Fault Modeling
Dataset
SNN Accelerator Configuration
Analyze the SNN Tolerance against Soft Errors
Study the impact of faulty synapses
Study the impact of faulty neurons
Study the impact of faulty synapses & neurons
SNN Fault Tolerance Characteristics
Develop and Employ Bound-and-Protect (BnP) Techniques
Weight Bounding
safe range
Neuron Protection
if ($V_{mem}$ reset is faulty):
o_spikes ← 0
Design Efficient Hardware Support for BnP Techniques
Weight Bounding
BnP1
BnP2 & BnP3
Neuron Protection
Bound-and-Protect (BnP) Technique
Enhanced SNN Compute Engine

Figure 5.32: An overview of the SoftSNN methodology. The novel contributions are highlighted in blue.

### 5.4.5.1 SNN Fault Tolerance Analysis

*Our SoftSNN first performs SNN fault tolerance analysis by characterizing the behavior of a given SNN model under different soft error profiles for the underlying hardware*, which provides beneficial information for devising lightweight soft error mitigation techniques. To do this, we perform experimental case studies for a 400-neuron network with the MNIST dataset since its size enables fast exploration for SNN fault tolerance analysis. Here, each

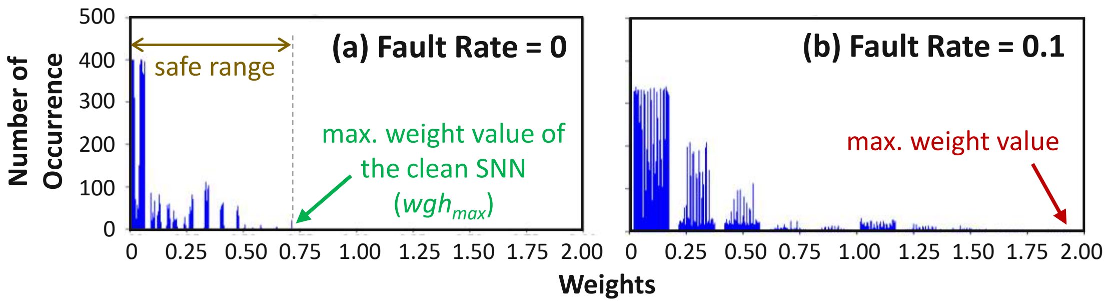


Figure 5.33: Soft errors can increase weight values, which may surpass the maximum weight value (i.e., $wgh_{max}$) from pre-trained SNN without soft errors (i.e., clean SNN).

SNN input has the same time range and coding for its spike train representation, and the employed STDP learning limits the weights in a certain range of positive values (e.g., $wgh = [0, 1]$), thereby making any workloads representative for the analysis. Following are the key steps of the experiments.

**Impact of faulty synapses:** We inject soft errors into the weight registers by randomly flipping the stored weight bits. The experimental results are shown in Figure 5.28 and Figure 5.33, from which we derive the following observations.

- Soft errors may increase or decrease weight values, and the increased ones have a more severe impact on accuracy since they trigger the neurons to generate spikes more frequently, thereby dominating classification.
- Increased weights may be recognized by employing the maximum weight value ($wgh_{max}$) of the pre-trained SNN without soft errors (i.e., clean SNN) as a threshold.

**Impact of faulty neurons:** We inject soft errors into the neuron hardware by randomly generating faulty neuron operations. The experimental results are shown in Figure 5.34(a), from which we obtain the following observations.

- Inference with faulty '$V_{mem}$ increase', '$V_{mem}$ leak', and 'spike generation' can achieve tolerable accuracy, as their faulty behavior does not make the neurons dominate classification, and the function for classifying the same input class may be substituted by other (non-faulty) neurons. Therefore, *these faulty neurons can still be employed for SNN processing.*
- Inference with faulty '$V_{mem}$ reset' can decrease the accuracy significantly, as this faulty behavior makes the neurons' membrane potential stays greater or equal to the threshold potential ($V_{mem} \geq V_{th}$), thereby generating (faulty) burst spikes and dominating classification. Therefore, *these faulty neurons should not be employed for SNN processing.*

**Impact of faulty synapses and neurons:** We inject soft errors by randomly flipping bits in the weight registers and generating faulty neuron operations. The experimental results in Figure 5.34(b) show that the faulty compute engine can severely decrease accuracy, thereby emphasizing the importance of soft error mitigation.

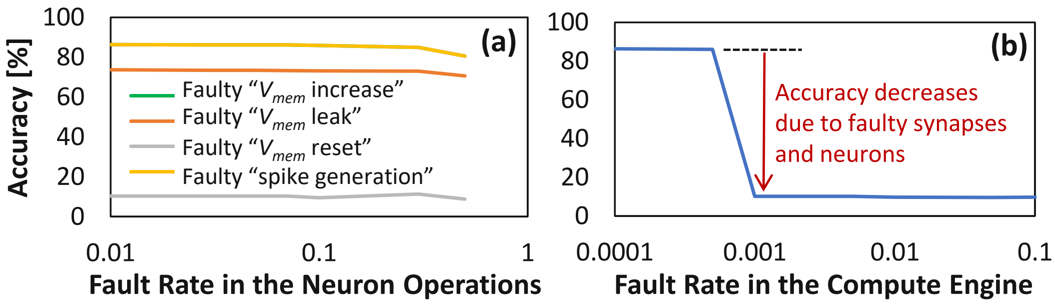


Figure 5.34: The impact of (a) faulty neuron operations, and (b) faulty weight registers and neuron operations, on accuracy.

### 5.4.5.2 Our Bound-and-Protect (BnP) Techniques

To detect and mitigate soft errors at run time, *we develop the Bound-and-Protect (BnP) technique, which bounds the weight values within a safe range that does not make neurons hyperactive (i.e., weight bounding), and protects the neurons from performing faulty operations that can significantly decrease accuracy (i.e., neuron protection).*

**Weight Bounding:** This mechanism clips the weight values that are greater or equal to the weight threshold ($wgh \geq wgh_{th}$), and replaces them with a pre-defined value ($wgh_{def}$), as stated in Equation 5.1. Hence, each weight has the bounding value ($wgh_b$) that does not trigger neurons' hyper-activity. To define $wgh_{th}$, we leverage the SNN fault tolerance characteristics from Section 5.4.5.1. We consider the range of weight values from the pre-trained SNN without soft errors (clean SNN) as the *safe range*, and employ its maximum value as the weight threshold ($wgh_{th} = wgh_{max}$), as shown in Figure 5.33(a).

$$wgh_b = \begin{cases} wgh_{def} & \text{if } wgh \geq wgh_{th} \\ wgh & \text{if otherwise} \end{cases} \tag{5.1}$$

**Neuron Protection:** This mechanism focuses on mitigating faulty '$V_{mem}$ reset' operations, as suggested by analysis in Section 5.4.5.1. We detect the faulty '$V_{mem}$ reset' operation in each neuron by monitoring the comparison output of $V_{mem} \geq V_{th}$. If the output is 'true' for multiple clock cycles (e.g., $\geq$ 2 clock cycles in this work), then it indicates that the '$V_{mem}$ reset' operation does not work properly. To efficiently address this, we disable the spike generation to prevent the corresponding neuron from generating burst spikes.

*We leverage these mechanisms for devising three variants of BnP techniques (i.e., BnP1, BnP2, and BnP3)*, which provide trade-offs in terms of accuracy, latency, and energy for soft error mitigation.

- **BnP1 Technique:** It replaces the weights that are greater or equal to the $wgh_{th}$ with zero. Therefore, the BnP1 can be stated as Equation 5.1 with $wgh_{def} = 0$.
- **BnP2 Technique:** It replaces the weights that are greater or equal to the $wgh_{th}$ with the maximum weight value from clean SNN ($wgh_{max}$). Therefore, the BnP2 can be stated as Equation 5.1 with $wgh_{def} = wgh_{max}$.

- **BnP3 Technique:** It replaces the weights that are greater or equal to the $wgh_{th}$ with a highly probable value from the weight distribution of clean SNN ($wgh_{hp}$). Therefore, the BnP3 can be stated as Equation 5.1 with $wgh_{def} = wgh_{hp}$.

- **For All BnP Techniques:** We continuously monitor the neuronal dynamics, and if the faulty '$V_{mem}$ reset' operation occurs, we disable the respective spike generation.

#### 5.4.5.3 Our Hardware Support for BnP Techniques

Performing the BnP techniques on the SNN accelerators at run time is challenging, as these accelerators typically have fixed dataflows. Therefore, *we propose lightweight self-healing hardware enhancements to support the deployment of our BnP techniques on the SNN accelerators without changing the dataflows*, as described in the following.

**Synapse Part:** The synapse enhancements aim at enabling the weight bounding, and they depend on the type of BnP technique.

- In the case of the **BnP1 Technique**, we add (1) a radiation-hardened register for storing the weight threshold $wgh_{th}$, which is used for all synapses in the compute engine, and (2) the hardened combinational logic units for performing a comparison and multiplexing in each synapse; see Figure 5.35(a).

- In case of the **BnP2 and BnP3 Techniques**, we add (1) two radiation-hardened registers for storing the weight threshold $wgh_{th}$ and the pre-defined weight value $wgh_{def}$ respectively, which are used for all synapses in the compute engine, and (2) the hardened combinational logic units for performing a comparison and multiplexing in each synapse; see Figure 5.35(b).

**Neuron Part:** To recognize faulty '$V_{mem}$ reset' operation, we monitor the comparison output of $V_{mem} \geq V_{th}$. If the output is 'true' for $\geq 2$ clock cycles, then the '$V_{mem}$ reset' operation is faulty. To ensure that such faulty operations do not result in burst spikes, we add an AND logic and a multiplexer to leverage the current and upcoming outputs for determining if the neuron should generate a spike in the next cycle, as shown in Figure 5.35(c).

**Radiation Hardening:** Since our hardware enhancements can also be affected by soft errors, we consider radiation-hardened components for all the new hardware extensions to make them resistant to high-energy particle strikes. To do this, the hardening techniques that improve the fabrication process (e.g., re-sizing transistor and insulating substrates [GJKC09, HJ19]) are employed. Here, we only need to harden the additional components, since they will provide correct values which can replace the corrupted bits in the subsequent circuits. Hence, the overhead of the hardening process is relatively low as compared to the full architecture of the SNN hardware, and will be discussed further in Section 5.4.7.2 (i.e., area overhead).

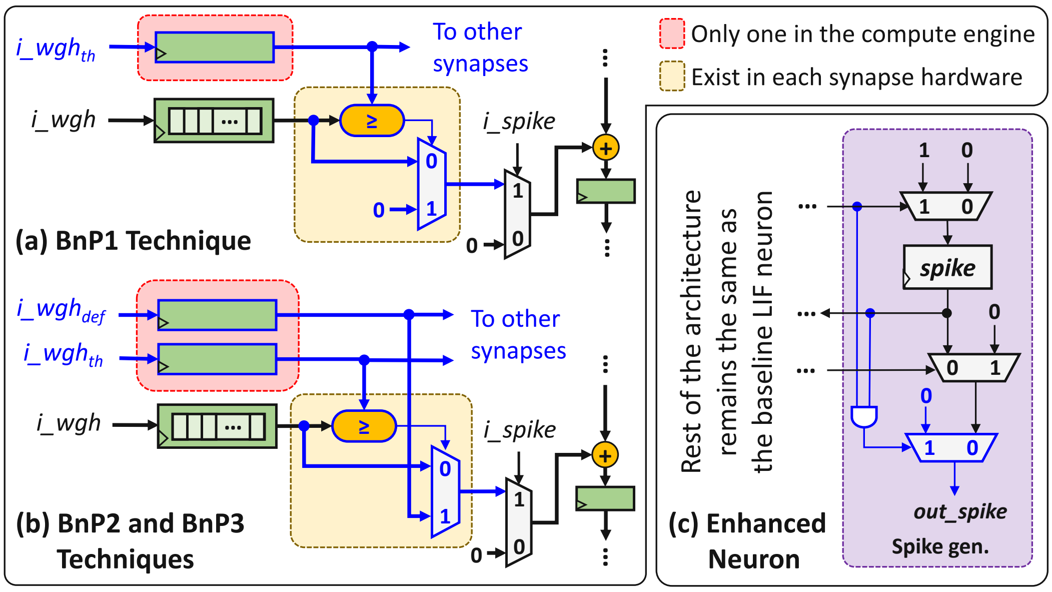

Only one in the compute engine
Exist in each synapse hardware
$i\_wgh_{th}$
i_wgh
To other synapses
i_spike
(a) BnP1 Technique
$i\_wgh_{def}$
$i\_wgh_{th}$
(b) BnP2 and BnP3 Techniques
Rest of the architecture remains the same as the baseline LIF neuron
spike
out_spike
Spike gen.
(c) Enhanced Neuron

Figure 5.35: The proposed synapse architectures for (a) BnP1, and (b) BnP2 and BnP3 techniques. (c) The proposed neuron architecture for mitigating faulty '$V_{mem}$ reset' operation. All circuit enhancements are highlighted in blue. Note, the reference of SNN accelerator architecture is shown in Figures 2.14-2.15.

### 5.4.6 Evaluation Methodology

We deploy the experimental setup presented in Figure 5.36 for evaluating our SoftSNN methodology, and adopt the same evaluation conditions as employed widely by the SNN community. We use the FC-based network as shown in Figure 4.8(a) with a different number of neurons for evaluating the generality of our SoftSNN methodology. For simplicity, we refer a network with $i$-number of neurons to as N$i$. We employ the MNIST and Fashion MNIST datasets as workloads. For comparison partners, we consider (1) the SNN without mitigation (i.e., No Mitigation), and (2) the SNN with 3x redundant executions and majority voting (i.e., Re-execution in TMR mode).

**Accuracy Evaluation:** We use a Python-based framework [HSK$^{+}$18], which run on multi-GPU machines (i.e., Nvidia RTX 2080 Ti), while considering the SNN accelerator architecture in Figures 2.14-2.15.

**Hardware Evaluations:** We implement the SNN compute engine illustrated in Figure 2.14(b) and Figure 2.15 with a 256x256 synapse crossbar. Its timing, power, and area are obtained through hardware synthesis using the Cadence Genus with a 65 nm CMOS technology library. We estimate the latency of compute engine for both with and without hardware enhancements, by leveraging the computation time for an inference of a single input sample. Afterward, we leverage the obtained latency and power to estimate the energy consumption of the compute engine.

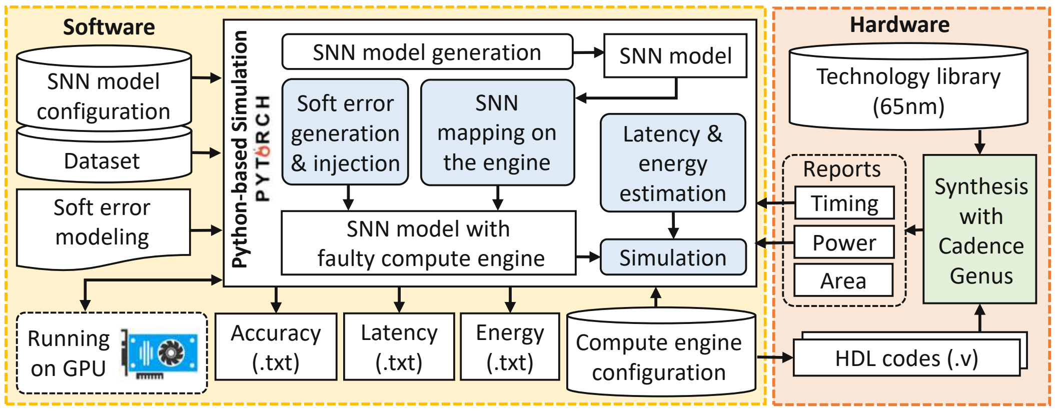


Figure 5.36: The experimental setup and tools flow.

## 5.4.7 Results and Discussion

### 5.4.7.1 Accuracy Comparisons

Figure 5.37 presents the experimental results for the accuracy of different mitigation techniques across various test scenarios. The re-execution technique can achieve high accuracy as shown by ❶, since it employs redundant executions to ensure consistent outputs, which indicates that the executions are minimally affected by soft errors. Meanwhile, our BnP techniques (BnP1, BnP2, and BnP3) achieve comparable accuracy to the re-execution, and significantly improve accuracy compared to the SNN without mitigation, i.e., by up to 80% and 47% for the MNIST and the Fashion MNIST, respectively; see ❷. The reason is that, our techniques employ safe weight values and safe neuron operations to avoid faulty neuronal dynamics that can significantly decrease accuracy. We observe that the BnP2 has slightly lower accuracy compared to the BnP1 and the BnP3, as it employs $wgh_{max}$ as $wgh_{def}$, whose values have low probability in the weight distribution of the clean SNN; see ❸. Hence, the generated neuronal dynamics do not closely match the neuronal dynamics of the clean SNN. We also observe that the BnP1 and the BnP3 have comparable accuracy, as their $wgh_{def}$ are relatively close to each other in the weight distribution of clean SNN; see ❹. Hence, the neuronal dynamics of the BnP1 and the BnP3 are similar. However, the BnP3 has better applicability for diverse applications than the BnP1, since the $wgh_{def}$ in the BnP3 can be updated for different weight distributions. These results show that *our BnP techniques are effective for mitigating soft errors in the SNN compute engine at run time without re-execution.*

### 5.4.7.2 Latency, Energy Consumption, and Area Overheads

Besides accuracy, we also evaluate the design overheads (i.e., latency, energy consumption, and area) incurred by different mitigation techniques.

**Latency:** Experimental results for latency are shown in Figure 5.38(a). We observe that the re-execution technique incurs ~3x latency as compared to the SNN without mitigation, as it employs redundant executions for loading parameters on the compute engine and performing SNN operations. Meanwhile, our BnP techniques only incur less

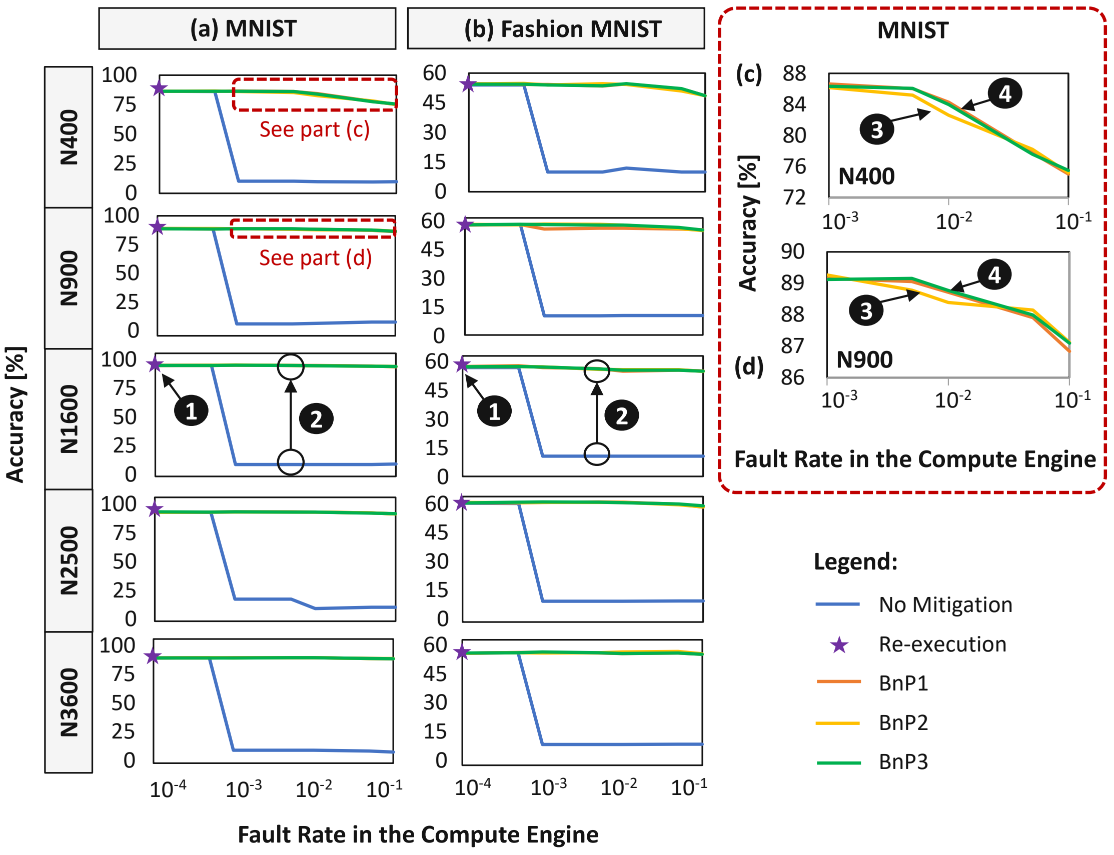


Figure 5.37: Results on accuracy across different mitigation techniques, network sizes, fault rates, and workloads: (a) MNIST and (b) Fashion MNIST. Detailed accuracy profiles on MNIST are shown for (c) N400 and (d) N900.

than 1.06x latency as compared to the SNN without mitigation, and reduce latency by up to 3x as compared to the re-execution, due to our efficient hardware modifications that minimally affect the latency (i.e., a small number of registers and combinational logic units without noticeably affecting the critical path). In this manner, our BnP-enhanced compute engine preserves the existing processing dataflow, and enables reliable SNN executions in the presence of soft errors for latency-constrained (real-time) applications.

**Energy Consumption:** Experimental results for energy consumption are shown in Figure 5.38(b). We observe that, the re-execution technique incurs 3x energy consumption overhead as compared to the SNN without mitigation, due to its redundant executions. Meanwhile, our BnP techniques incur less than 1.6x energy consumption when compared to the SNN without mitigation, and reduce the energy consumption by up to 2.3x as compared to the re-execution. Note, compared to the original hardware executing SNN without mitigation, the slight increase in the energy consumption of our BnP-enhanced hardware is due to the additional hardware components to enable reliable SNN execution without incurring noticeable latency/performance overheads. Moreover,

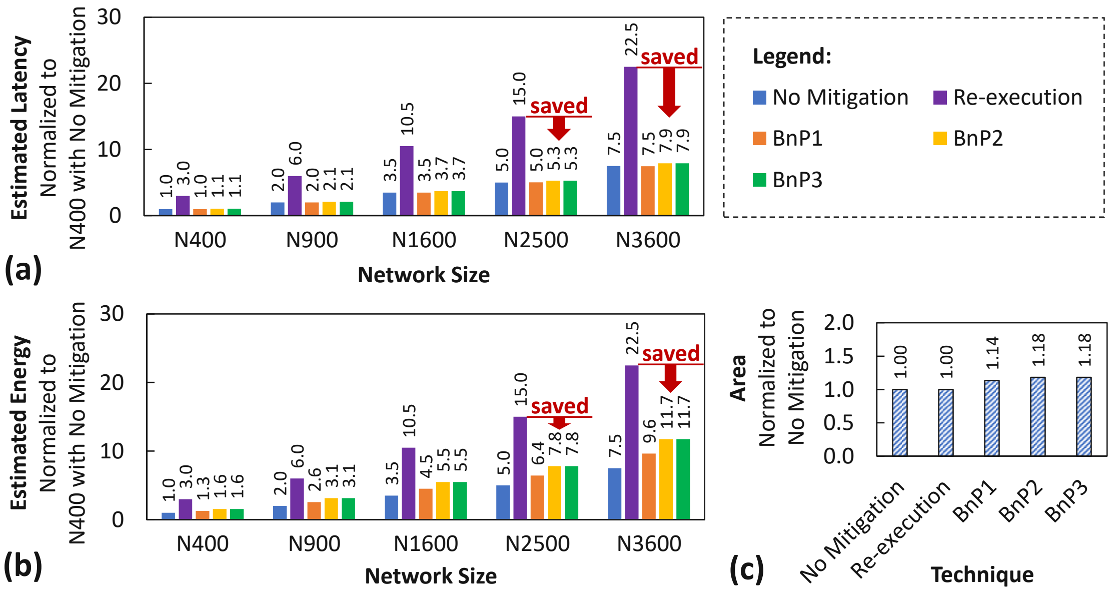


Figure 5.38: Comparisons across different techniques and network sizes on (a) latency and (b) energy for an inference of a single input, and (c) area. Results for the MNIST and the Fashion MNIST are similar, as these workloads have the same input dimension.

since redundant executions are completely avoided, our techniques substantially optimize energy consumption as compared to the re-execution-based mitigation technique.

**Area:** Experimental results for the area are shown in Figure 5.38(c). We observe that our BnP-enhanced compute engine incurs tolerable area overhead (i.e., 14% for the BnP1, and 18% for the BnP2 and the BnP3) as compared to the compute engine without enhancements. These area overheads mainly come from additional components in synapses, as the synapse crossbar dominates the area of compute engine. Furthermore, the area overhead also represents the cost of the new radiation-hardened components to ensure reliable SNN execution. Note, we only need to harden the additional components for providing correct bits to the subsequent circuits, thereby correcting the corrupted bits and ensuring reliable executions in the respective circuits with low overhead.

In summary, all these results show that *our BnP techniques effectively mitigate soft errors in the SNN compute engine, while significantly reducing the latency and the energy of SNN executions as compared to the re-execution-based mitigation techniques.*

#### 5.4.7.3 Algorithm Complexity

The algorithm complexity of SoftSNN can be derived from its mapping solution. Here, the weight bounding operations are performed to limit the weight values on the faulty synaptic registers, and the neuron protection operations are performed to avoid harmful neuron behavior on the faulty neurons. Such weight bounding and neuron protection operations have constant time and space complexity. Therefore, the algorithm complexity of SoftSNN mapping solution can be stated as $O(n)$ for time complexity with $n$ denotes the number of SNN parameters (i.e., weight and neuron parameters), and $O(1)$ for space complexity. Here, optimal solutions that mitigate soft errors to maintain accuracy without incurring overheads are not feasible, as SoftSNN employs faulty synaptic registers and faulty neurons to allow errors/approximations in computation that lead to a slightly reduced yet acceptable accuracy, while maintaining throughput and incurring minimum overheads (e.g., in latency, area, and energy consumption).

### 5.4.8 Summary of SoftSNN Methodology

We propose the novel SoftSNN methodology for mitigating soft errors in the compute engine of SNN accelerators/chips without re-execution. Our SoftSNN analyzes the SNN characteristics under soft errors, performs weight bounding and neuron protection, and devises efficient hardware enhancements to enable the proposed techniques. The results show that, our SoftSNN maintains high accuracy while reducing latency and energy consumption, as compared to the re-execution-based mitigation techniques, thereby enabling reliable SNN executions against soft errors for real-time and energy-efficient SNN-based systems and applications.

## 5.5 Summary of Fault-Tolerant SNN Systems

This chapter discusses our novel methodology for enabling reliable SNN systems. It systematically addresses the HW-induced faults in the (off-chip and on-chip) memories and the compute engine of SNN neuromorphic accelerators using our proposed HW/SW-level fault mitigation techniques. Specifically, it aims at mitigating faults in the DRAM and weight buffer parts through fault-aware mapping if the training dataset is not fully available, and through fault-aware training-and-mapping if the training dataset is fully available. Besides faults in memories, our methodology also mitigates faults in the SNN compute engine. It employs fault-aware mapping and lightweight HW enhancements to safely map SNN weights and operations to the faulty SNN compute engine, thereby minimizing the negative impact of permanent faults on the accuracy while maintaining the processing throughput. Our methodology also employs weight bounding, neuron protection, and lightweight HW enhancements to identify anomaly behavior of SNN operations due to soft errors in the SNN compute engine, then efficiently mitigate them without re-execution. In this manner, our SNN fault-tolerant techniques can be coupled with our SNN optimization techniques in Chapter 4 to enable energy-efficient and reliable SNN processing for many AI applications with tightly-constrained memory and power/energy budgets (e.g., Edge-AI and Smart CPS).

CHAPTER 6

# Conclusion and Future Outlook

## 6.1 Thesis Summary

Bringing the powerful capabilities of NN algorithms (DNNs and SNNs) for solving data analytic tasks in resource- and energy-constrained embedded applications is highly desired since it has huge potential to provide better quality of services, higher efficiency, lower latency, better security, and better privacy, thereby improving the productivity of human life. However, embedded implementation of NN algorithms is a challenging task due to (1) memory- and compute-intensive nature of NNs that may violate the memory and energy constraints, and (2) possible HW-induced faults that may degrade the accuracy. Therefore, energy efficiency and fault tolerance aspects are considered important as they ensure that the NN processing can be performed in the given computing platforms to produce reliable outputs. Toward this, *the research goal of this thesis is to achieve high energy efficiency and high fault tolerance in NN-based systems.*

This research goal imposes scientific research challenges. In DNN systems, the challenges are about leveraging the DRAM access characteristics and the dataflow of DNN processing to minimize the DRAM access energy, which dominates the DNN systems' total energy. Meanwhile, in SNN systems, the challenges are about optimizing the memory and energy requirements of SNN processing, devising a lightweight unsupervised learning mechanism considering dynamic and non-dynamic environments, and developing cost-effective techniques for mitigating HW-induced faults, such as approximation errors, permanent faults, and soft errors.

To systematically address these research challenges, this thesis breaks down the research goal into several key research objectives, as summarized in the following.

- *Memory access energy optimization for DNN systems:* Previous works observe that optimizing the data partitioning and scheduling is effective to reduce the DRAM accesses and improve the energy of DNN accelerators. However, they do not optimize the redundant DRAM accesses for the overlapping data partition and the DRAM energy-per-access. Therefore, we aim at optimizing the redundant DRAM accesses for the overlapping data partition and the DRAM energy-per-access of the DNN systems.

- *Memory access energy optimization for SNN systems:* Previous works employ pruning, quantization, and data bundling to optimize the memory access energy. However, they incur high overheads for data encoding, may suffer from information loss, and do not optimize the inhibitory operations which incur considerable memory and energy. Therefore, we aim at minimizing the memory requirement of SNNs by optimizing the inhibitory operations, employing quantization, while improving the STDP-based learning mechanism.

- *HW-level optimization for SNN systems:* Previous works typically employ HW accelerators to improve the performance efficiency of SNN processing. However, this solution incurs high design time. Therefore, we aim at employing approximate hardware (e.g., DRAM) in SNN HW accelerators to substantially reduce the operational power/energy of SNN systems, while meeting the design constraints (e.g., accuracy).

- *Unsupervised continual learning for SNN systems:* Previous works typically incur high memory and energy requirements due to their large model (e.g., inhibitory neurons and additional components) and complex exponential computations. Therefore, we aim at understanding the impact of network components (e.g., inhibitory layer) and SNN parameters (e.g., weight decay) on the accuracy, and then leverage this information to determine how the learning process should be performed efficiently.

- *Fault mitigation techniques for SNN systems:* Previous works still focus on studying different types of faults in SNNs and the impact of random faults on accuracy, without considering detailed fault models and the underlying SNN HW architecture. Therefore, we aim at understanding the impact of HW-induced faults in SNN systems, and then leverage these studies to develop cost-effective fault-mitigation techniques.

This thesis fulfills the research objectives through several novel scientific contributions, as summarized in the following.

- **HW/SW-level DRAM optimization for energy-efficient DNN systems:** We perform DRAM access optimization through an exploration to find data partitioning and scheduling that achieve the minimum number of DRAM accesses while minimizing redundant DRAM accesses for the overlapping data partition. We also develop a generalized DRAM data mapping policy to further optimize the DRAM energy-per-access for any given DRAM architecture in DNN systems (e.g., commodity DRAMs or new DRAM architectures from the literature).

- **HW/SW-level design and optimization for energy-efficient SNN systems:** We optimize SNN operations through the removal of an inhibitory layer and simplification of the weight update mechanism. Then, weight quantization is performed to reduce the memory footprint. We also exploit approximate DRAM to significantly reduce the operational power/energy of SNN HW accelerators, while minimizing the impact of approximation errors on accuracy through fault-aware training. To make the SNN systems adaptive to different environments (i.e., dynamic or non-dynamic), we use the spiking activity to develop an adaptive unsupervised learning mechanism and parameter enhancements (e.g., weight decay). Here, memory and energy estimation can be utilized to quickly find an SNN model that meets the memory and energy budgets, thereby providing appropriate SNN model for the given operational settings.

- **Cost-effective HW/SW-level fault tolerance for SNN systems:** We mitigate faults in the off-chip and on-chip memories through fault-aware mapping, especially when the training set is not fully available (e.g., due to IP and privacy reasons), and may employ fault-aware training-and-mapping if the training set is fully available to further improve the SNN resilience. Then, we mitigate permanent faults in the compute engine of SNN accelerators through fault-aware mapping techniques to safely map weight bits to the faulty synapses and selectively utilize faulty neurons without any retraining. Here, lightweight HW enhancements are devised to accommodate data transformation due to the mapping technique. We also mitigate soft errors in the compute engine of SNN accelerators through weight bounding and neuron protection using lightweight HW circuits to ensure that the weight values and neuron behavior do not lead to significant accuracy degradation.

In summary, *this thesis proposes a design methodology that employs novel techniques to enable high energy efficiency and high fault tolerance for NN-based systems*, thereby making it suitable for diverse resource- and energy-constrained embedded applications.

## 6.2 Future Works

The field of NNs (DNNs and SNNs) is progressing fast. New network models, hardware platforms (e.g., GPUs, accelerators), application use-cases for NNs, and techniques for improving the performance and efficiency of NNs are proposed and published on a daily basis. Therefore, several opportunities and potential directions of future works can be explored based on the techniques proposed in this thesis, as discussed in the following.

- **Enhance the techniques in this thesis for new network models and hardware architectures:** The development of DNNs and SNNs is typically driven by the target of achieving higher accuracy, hence leading to the proliferation of new network models and hardware architectures. Therefore, a possible direction is to explore the memory and energy requirements as well as the resilience of new DNN and SNN models considering new underlying hardware architectures and different types of faults. Here, the design, optimization, and fault-mitigation techniques in this thesis can be enhanced and fine-tuned to properly improve the energy efficiency and fault tolerance of newly developed NN-based systems.

- **Low-cost techniques for mitigating the negative impact of device aging:** Aging in nano-scale electronic devices happens because of physical phenomena such as TDDB, HCI, BTI, and EM. Aging causes timing errors in the early stages of occurrence, and later it can transform into permanent faults. Therefore, a possible direction is to explore low-cost techniques for mitigating the negative impact of aging before it transforms into permanent faults.
- **Developing SNN models that can learn complex features efficiently:** The accuracy of SNNs with bio-plausible learning rules (e.g., STDP) for complex input features is usually lower than DNNs, hence may limit the applicability of SNNs. However, SNNs bear the potential to achieve ultra-low power/energy for both learning and inference due to their sparse spike-based computations. SNNs also have the potential to enable smart systems with tight memory and energy budgets to adapt to changing operational environments due to their unsupervised learning capabilities. Therefore, a possible direction is to develop new SNN models with bio-plausible learning rules that can achieve high accuracy for complex input features considering diverse operational environments.
- **Developing highly efficient and fault-tolerant SNN accelerators for Edge-AI:** Edge-AI applications require ultra-low-power/energy computing platforms to run their workloads. Toward this, SNNs have the potential to meet the memory and energy requirements of such applications. However, the existing SNN accelerators for Edge-AI only support limited networks and limited quality of online learning. Therefore, a possible direction is to develop SNN accelerators that incorporate efficient dataflows for diverse spiking networks, ultra-low-power memory and compute units, efficient online learning mechanisms, and lightweight fault mitigation mechanisms. Here, an experimental platform that can apply different voltage settings and capture the map of approximation-induced errors for off-chip and on-chip memories is required, as it helps in characterizing memories for developing the accurate models of SNN accelerators.
- **Defending SNN accelerators from adversarial attacks for Edge-AI:** Edge-AI applications require secured computing platforms to run their workloads. However, the existing SNN accelerators for Edge-AI have not considered the security aspect in their designs, including the adversarial attacks and their defense mechanisms. Therefore, a possible direction is to study the impact of different possible adversarial attacks on SNN accelerators (e.g., Rowhammer attacks [KPY+20]), and then develop cost-effective defense mechanisms.
- **Implementation of NNs on Processing-in-Memory (PIM):** Technological advancements like PIM may open new opportunities and avenues for realizing highly efficient NN-based systems. In PIM systems, the dot-product operations are performed in the analog domain (e.g., using ReRAM devices), which suffers from several issues such as limited precision and IR drop, hence making it difficult to realize reliable HW accelerators (for both DNNs and SNNs). Therefore, a possible direction is to improve both hardware and software parts, including by leveraging the knowledge and insights learned from research in this thesis.

# List of Figures

# List of Tables

# Bibliography

[AAH$^+$20] Hazoor Ahmad, Tabasher Arif, Muhammad Abdullah Hanif, Rehan Hafiz, and Muhammad Shafique. SuperSlash: A unified design space exploration and model compression methodology for design of deep learning accelerators with reduced off-chip memory access volume. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):4191–4204, 2020.

[AAK$^+$21] Amogh Agrawal, Mustafa Ali, Minsuk Koo, Nitin Rathi, Akhilesh Jaiswal, and Kaushik Roy. IMPULSE: A 65-nm digital compute-in-memory macro with fused weights and membrane potential for spike-based sequential learning tasks. *IEEE Solid-State Circuits Letters (LSSC)*, 4:137–140, 2021.

[ABAR17] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha. Deep learning algorithm for autonomous driving using googlenet. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 89–96, 2017.

[Abb99] Larry F Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain research bulletin*, 50(5-6):303–304, 1999.

[ADJ$^+$17] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 382–394, 2017.

[AES21] Sarah Ali El Sayed. *Fault Tolerance in Hardware Spiking Neural Networks.* Theses, Sorbonne Université, October 2021.

[AHMK21] Daniel Auge, Julian Hille, Etienne Mueller, and Alois Knoll. A survey of encoding techniques for signal processing in spiking neural networks. *Neural Processing Letters*, 53(6):4693–4710, 2021.

[AHS17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3), February 2017.

[AJH+16] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *43rd International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[AMY+23] Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R Young, Frank Liu, and Jeffrey S Vetter. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems (Memori)*, 4:100022, 2023.

[Ant19] Gary Anthes. Lifelong learning in artificial neural networks. *Commun. ACM*, 62(6):13–15, May 2019.

[AR16] J. M. Allred and K. Roy. Unsupervised incremental stdp learning using forced firing of dormant or idle neurons. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 2492–2499, 2016.

[AR20] Jason M. Allred and Kaushik Roy. Controlled forgetting: Targeted stimulation and dopaminergic plasticity modulation for unsupervised lifelong learning in spiking neural networks. *Frontiers in Neuroscience (FNINS)*, 14:7, 2020.

[ASC+15] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. TrueNorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems (TCAD)*, 34(10):1537–1557, October 2015.

[ATB+17] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey McKinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, Jeff Kusnitz, Michael Debole, Steve Esser, Tobi Delbruck, Myron Flickner, and Dharmendra Modha. A low power, fully event-based gesture recognition system. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7388–7397, July 2017.

[AZH+23] Hassen Aziza, Cristian Zambelli, Said Hamdioui, Sumit Diware, Rajendra Bishnoi, and Anteneh Gebregiorgis. On the reliability of rram-based neural networks. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–8, 2023.

[B+09] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[ban] bankmycell. How many smartphones are in the world? `https://www.bankmycell.com/blog/how-many-phones-are-in-the-world`. Accessed: 2023-01-01.

[Bau05] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 5(3):305–316, 2005.

[BBD19] Naveed Khan Baloch, Muhammad Iram Baig, and Masoud Daneshtalab. Defender: A low overhead and efficient fault-tolerant mechanism for reliable on-chip router. *IEEE Access*, 7:142843–142854, 2019.

[BDFZ22] Arindam Basu, Lei Deng, Charlotte Frenkel, and Xueyong Zhang. Spiking neural network integrated circuits: A review of trends and future directions. In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2022.

[BG05] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology*, 94(5):3637–3642, 2005.

[BGM$^+$14] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[BKL02] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17–37, 2002.

[BKM$^+$17] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2), 2017.

[BNH$^+$13] Stephen Brink, Stephen Nease, Paul Hasler, Shubha Ramakrishnan, Richard Wunderlich, Arindam Basu, and Brian Degnan. A learning-enabled neuron array IC based upon transistor channel models of biological phenomena. *IEEE Transactions on Biomedical Circuits and Systems (TBCAS)*, 7(1):71–81, 2013.

[BSF07] Joseph M. Brader, Walter Senn, and Stefano Fusi. Learning real-world stimuli in a neural network with spike-driven synaptic dynamics. *Neural Computation*, 19(11):2881–2912, 2007.

[CBM$^+$20a] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020.

[CBM$^+$20b] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7), 2020.

[CBM$^+$20c] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7):113, 2020.

[CCK15] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113:54–66, 2015.

[CDS$^+$14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[CGF22] Qinyu Chen, Chang Gao, and Yuxiang Fu. Cerebron: A reconfigurable architecture for spatiotemporal sparse spiking neural networks. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 30(10):1425–1437, 2022.

[CH84] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[Cha14] Karthik Chandrasekar. *High-level power estimation and optimization of DRAMs*. PhD thesis, TU Delft, 2014.

[CKES17] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[CL18] Zhiyuan Chen and Bing Liu. Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–207, 2018.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.

[CMA+13] Andrew S. Cassidy, Paul Merolla, John V. Arthur, Steve K. Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M. Wong, Vitaly Feldman, Arnon Amir, Daniel Ben-Dayan Rubin, Filipp Akopyan, Emmett McQuinn, William P. Risk, and Dharmendra S. Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2013.

[CPS17] Anupam Chattopadhyay, Alok Prakash, and Muhammad Shafique. Secure cyber-physical systems: Current trends, tools and open research problems. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1104–1109, 2017.

[CR19] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[CSLC21] Fan Chen, Linghao Song, Hai Li, and Yiran Chen. MARVEL: A vertical resistive accelerator for low-power deep learning inference in monolithic 3D. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1240–1245. IEEE, 2021.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv preprint arXiv:1511.07289*, 2015.

[CYES18a] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Understanding the limitations of existing energy-efficient design approaches for deep neural networks. In *2018 Proceedings of SysML Conference (SysML)*, 2018.

[CYES18b] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Understanding the limitations of existing energy-efficient design approaches for deep neural networks. *Energy*, 2(L1):L3, 2018.

[CYES19] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, 9(2):292–308, 2019.

[CYG+17a] Kevin K. Chang, A. Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), June 2017.

[CYG+17b] Kevin K. Chang, A. Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding reduced-voltage operation in modern DRAM devices: Experimental characterization, analysis, and mechanisms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 1(1), jun 2017.

[DBDRC+15] Zidong Du, Daniel D. Ben-Dayan Rubin, Yunji Chen, Liqiang He, Tianshi Chen, Lei Zhang, Chengyong Wu, and Olivier Temam. Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches. In *48th International Symposium on Microarchitecture (MICRO)*, page 494–507, 2015.

[DC15] Peter Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9:99, 2015.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[DFC+15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[DFG+11] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *8th ACM International Conference on Autonomic Computing (ICAC)*, page 31–40, 2011.

[DLH+20] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[DMB+12a] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 143–154, 2012.

[DMB+12b] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. MultiScale: Memory system DVFS with multiple memory controllers. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, page 297–302, 2012.

[DMR+11] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: Active low-power modes for main memory. In *The 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 225–238, 2011.

[DSL+18] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018.

[DY14] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and trends® in Signal Processing*, 7(3–4):197–387, 2014.

[EBDB20] Hammouda Elbez, Kamel Benhaoua, Philippe Devienne, and Pierre Boulet. Progressive compression and weight reinforcement for spiking neural networks. *HAL*, hal-02737057, Jun 2020. hal-02737057.

[EK86] G Bard Ermentrout and Nancy Kopell. Parabolic bursting in an excitable system coupled with a slow oscillation. *SIAM journal on applied mathematics*, 46(2):233–253, 1986.

[ESSP+20] Sarah A. El-Sayed, Theofilos Spyrou, Antonios Pavlidis, Engin Afacan, Luis A. Camuñas-Mesa, Bernabé Linares-Barranco, and Haralampos-G. Stratigopoulos. Spiking neuron hardware-level fault modeling. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4, 2020.

[FAB+00] Stefano Fusi, Mario Annunziato, Davide Badoni, Andrea Salamon, and Daniel J Amit. Spike-driven synaptic plasticity: Theory, simulation, VLSI implementation. *Neural Computation*, 12(10):2227–2258, 2000.

[Fal19] Pierre Falez. *Improving Spiking Neural Networks Trained with Spike Timing Dependent Plasticity for Image Recognition.* Theses, Université de Lille, October 2019.

[FCT22] Shu-Kai S. Fan, Chun-Wei Cheng, and Du-Ming Tsai. Fault diagnosis of wafer acceptance test and chip probing between front-end-of-line and back-end-of-line processes. *IEEE Transactions on Automation Science and Engineering*, 19(4):3068–3082, 2022.

[FFW22] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. SmartApprox: Learning-based configuration of approximate memories for energy-efficient execution. *Sustainable Computing: Informatics and Systems*, 34:100701, 2022.

[Fit61] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

[FLB19] Charlotte Frenkel, Jean-Didier Legat, and David Bol. MorphIC: A 65-nm 738k-synapse/mm$^2$ quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning. *IEEE Transactions on Biomedical Circuits and Systems (TBCAS)*, 13(5):999–1010, 2019.

[FLLB19] Charlotte Frenkel, Martin Lefebvre, Jean-Didier Legat, and David Bol. A 0.086-mm$^2$ 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm cmos. *IEEE Transactions on Biomedical Circuits and Systems (TBCAS)*, 13(1):145–158, Feb 2019.

[Flo12] Răzvan V. Florian. The Chronotron: A neuron that learns to fire temporally precise spike patterns. *PLOS ONE*, 7(8):1–27, 08 2012.

[Fri05] Pascal Fries. A mechanism for cognitive dynamics: Neuronal communication through neuronal coherence. *Trends in Cognitive Sciences*, 9(10):474–480, 2005.

[FTHVVB03] Nicolas Fourcaud-Trocmé, David Hansel, Carl Van Vreeswijk, and Nicolas Brunel. How spike generation mechanisms determine the neuronal response to fluctuating inputs. *Journal of Neuroscience*, 23(37):11628–11640, 2003.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *32nd International Conference on on Machine Learning (ICML) - Volume 37*, page 1737–1746, 2015.

[GCTV19] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 151–165, 2019.

[GD03] Andrzej Granas and James Dugundji. *Fixed Point Theory*. Springer, 2003.

[Ger95] Wulfram Gerstner. Time structure of the activity in neural network models. *Physical Review E*, 51(1):738, 1995.

[GFES21] Wenzhe Guo, Mohammed E. Fouda, Ahmed M. Eltawil, and Khaled Nabil Salama. Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems. *Frontiers in Neuroscience (FNINS)*, 15, 2021.

[GFY$^+$20] Wenzhe Guo, Mohammed E. Fouda, Hasan Erdem Yantir, Ahmed M. Eltawil, and Khaled Nabil Salama. Unsupervised adaptive weight pruning for energy-efficient neuromorphic systems. *Frontiers in Neuroscience (FNINS)*, 14:1189, 2020.

[GGWB17] Edward Griffor, Christopher Greer, David Wollman, and Martin Burns. *Framework for Cyber-Physical Systems: Volume 1, Overview.* Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2017-06-26 2017.

[GJKC09] Rajesh Garg, Nikhil Jayakumar, Sunil P. Khatri, and Gwan S. Choi. Circuit-level design approaches for radiation-hard digital electronics. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 17(6):781–792, 2009.

[GK02] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity.* Cambridge university press, 2002.

[GKD$^+$] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC.

[GKNP14] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition.* Cambridge University Press, 2014.

[GKTB15] Shrikanth Ganapathy, Georgios Karakonstantis, Adam Teman, and Andreas Burg. Mitigating the impact of faults in unreliable memories for error-resilient applications. In *52nd Annual Design Automation Conference (DAC)*, 2015.

[GLH$^+$19] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying complex workload-dram interactions: An experimental study. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 3(3), December 2019.

[GLX$^+$17] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

[GMP$^+$11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. IMPACT: imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 409–414, 2011.

[GT98] Jacques Gautrais and Simon Thorpe. Rate coding versus temporal order coding: a theoretical approach. *Biosystems*, 48(1-3):57–65, 1998.

[GWG+22] Daniel Gerlinghoff, Zhehui Wang, Xiaozhe Gu, Rick Siow Mong Goh, and Tao Luo. A resource-efficient spiking neural network accelerator supporting emerging neural encoding. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 92–95, 2022.

[GYG+18] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. What your dram power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Measurement and Analysis of Computing Systems*, 2(3):38:1–38:41, Dec. 2018.

[HH52] Alan L. Hodgkin and Andrew F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[HJ19] Qiang Huang and Jin Jiang. An overview of radiation effects on electronic devices under severe accident conditions in NPPs, rad-hardened design techniques and simulation tools. *Progress in Nuclear Energy*, 114:105–120, 2019.

[HKP+18] Muhammad Abdullah Hanif, Faiq Khalid, Rachmad Vidya Wicaksana Putra, Semeen Rehman, and Muhammad Shafique. Robust machine learning systems: Reliability and security for deep neural networks. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 257–260, 2018.

[HKP+21] Muhammad Abdullah Hanif, Faiq Khalid, Rachmad Vidya Wicaksana Putra, Mohammad Taghi Teimoori, Florian Kriebel, Jeff Jun Zhang, Kang Liu, Semeen Rehman, Theocharis Theocharides, Alessandro Artusi, et al. Robust Computing for Machine Learning-based Systems. In *Dependable Embedded Systems*, pages 479–503. Springer, Cham, 2021.

[HLL+18] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *The European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[HLM+16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual Interna-*

*tional Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[HMD16] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations (ICLR)*, May 2016.

[HMLF20] Michael Hopkins, Mantas Mikaitis, Dave R Lester, and Steve Furber. Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philosophical Transactions of the Royal Society A (RSTA)*, 378(2166):20190052, 2020.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *Proceedings of the IEEE Int. Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb. 2014.

[HPT+18] Muhammad Abdullah Hanif, Rachmad Vidya Wicaksana Putra, Muhammad Tanvir, Rehan Hafiz, Semeen Rehman, and Muhammad Shafique. MPNA: A massively-parallel neural array accelerator with dataflow optimization for convolutional neural networks. *arXiv preprint arXiv:1810.12910*, 2018.

[HS20] Muhammad Abdullah Hanif and Muhammad Shafique. Salvagednn: Salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Philosophical Transactions of the Royal Society A (RSTA)*, 378(2164):20190164, 2020.

[HSK+18] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. BindsNET: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics (FNINF)*, 12:89, 2018.

[HSS+18] H. Hazan, D. Saunders, D. T. Sanghavi, H. Siegelmann, and R. Kozma. Unsupervised learning with self-organizing spiking neural networks. In *Proc. of the Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1–6, July 2018.

[HSS+19] Hananel Hazan, Daniel J. Saunders, Darpan T. Sanghavi, Hava Siegelmann, and Robert Kozma. Lattice map spiking neural networks (lm-snns) for clustering and classifying image data. *Annals of Mathematics and Artificial Intelligence*, Sep 2019.

[HVK+17] Hasan Hassan, Nandita Vijaykumar, Samira Khan, Saugata Ghose, Kevin Chang, Gennady Pekhimenko, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. SoftMC: A flexible and practical open-source infrastructure for enabling experimental DRAM studies. In *IEEE International Symposium* *on High Performance Computer Architecture (HPCA)*, pages 241–252, 2017.

[HYA+18] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *45th Annual International Symposium on Computer Architecture (ISCA)*, page 674–687, 2018.

[HYAK+20] Jawad Haj-Yahya, Mohammed Alser, Jeremie Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu. SysScale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 227–240, 2020.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IHM+16] Forrest N. Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning (ICML)*, volume 37, pages 448–456, 07–09 Jul 2015.

[Izh01] Eugene M. Izhikevich. Resonate-and-fire neurons. *Neural Networks*, 14(6-7):883–894, 2001.

[Izh03] Eeugene .M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks (TNN)*, 14(6):1569–1572, 2003.

[Izh04] Eeugene .M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks (TNN)*, 15(5):1063–1070, 2004.

[JED12] Standard JEDEC. JESD79-3F: DDR3 SDRAM Standard. *JEDEC Solid State Technology Association*, July 2012.

[JED15] Standard JEDEC. JESD209-3C: Low Power Double Data Rate 3 (LPDDR3) Standard. *JEDEC Solid State Technology Association*, August 2015.

[JED17a] Standard JEDEC. JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard. *JEDEC Solid State Technology Association*, March 2017.

[JED17b] Standard JEDEC. JESD79-4B: DDR4 SDRAM Standard. *JEDEC Solid State Technology Association*, June 2017.

[JKC$^+$18a] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[JKC$^+$18b] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[JLG04] Renaud Jolivet, Timothy J Lewis, and Wulfram Gerstner. Generalized integrate-and-fire models of neuronal activity approximate spike trains of a detailed model to a high degree of accuracy. *Journal of Neurophysiology*, 92(2):959–976, 2004.

[JWN10] Bruce Jacob, David Wang, and Spencer Ng. *Memory Systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[JYP$^+$17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*, page 1–12, 2017.

[KAY17] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. ZeNa: Zero-aware neural network accelerator. *IEEE Design & Test*, 35(1):39–46, 2017.

[KCW+22] Yisong Kuang, Xiaoxin Cui, Zilin Wang, Chenglong Zou, Yi Zhong, Kefei Liu, Zhenhui Dai, Dunshan Yu, Yuan Wang, and Ru Huang. ESSA: Design of a programmable efficient sparse spiking neural network accelerator. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 30(11):1631–1641, 2022.

[KGE11] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th International Conference on VLSI Design (VLSID)*, pages 346–351, 2011.

[KK98] I. Koren and Z. Koren. Defect tolerance in VLSI circuits: Techniques and yield analysis. *Proc. of the IEEE*, 86(9):1819–1838, 1998.

[KMN20] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic plasticity dynamics for deep continuous local learning (DECOLLE). *Frontiers in Neuroscience (FNINS)*, 14, 2020.

[KMZ19] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 821–834, New York, NY, USA, 2019. Association for Computing Machinery.

[KOY+19] Skanda Koppula, Lois Orosa, A. Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: enabling energy-efficient, high-performance deep neural network inference using approximate DRAM. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 166–181, 2019.

[KP06] Andrzej Kasiński and Filip Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal of Applied Mathematics and Computer Science*, 16(1):101–113, 2006.

[KPHM18] Jeremie Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. Solar-DRAM: Reducing DRAM access latency by exploiting the variation in local bitlines. In *IEEE 36th International Conference on Computer Design (ICCD)*, pages 282–291, 2018.

[KPMHB11] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling. *IEEE Micro*, 31(1):78–89, 2011.

[KPR+17] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences (PNAS)*, 114(13):3521–3526, 2017.

[KPY+20] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651, 2020.

[KRH+18] Florian Kriebel, Semeen Rehman, Muhammad Abdullah Hanif, Faiq Khalid, and Muhammad Shafique. Robustness for smart cyber physical systems and internet-of-things: From adaptive robustness methods to reliability and security for machine learning. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 581–586, 2018.

[Kri18] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[KSK18] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *23th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–475, 2018.

[KSL+12] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379, 2012.

[KSVR19] Sarada Krithivasan, Sanchari Sen, Swagath Venkataramani, and Anand Raghunathan. Dynamic spike bundling for energy-efficient spiking neural networks. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[KYM16] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters (LCA)*, 15(1):45–49, Jan 2016.

[LBBH98a] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBBH98b] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LDBK20] Jesus L. Lobo, Javier Del Ser, Albert Bifet, and Nikola Kasabov. Spiking neural networks and online learning: An overview and perspectives. *Neural Networks*, 121:88 – 100, 2020.

[LDT+23] Guoqi Li, Lei Deng, Huajing Tang, Gang Pan, Yonghong Tian, Kaushik Roy, and Wolfgang Maass. Brain Inspired Computing: A Systematic Survey and Future Trends. 1 2023.

[LJG+19] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules. *IEEE Transactions on Computers (TC)*, 68(11):1663–1677, 2019.

[LKD+16] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[LKJ+17] Sang-Woo Lee, Jin-Hwa Kim, Jaehyun Jun, Jung-Woo Ha, and Byoung-Tak Zhang. Overcoming catastrophic forgetting by incremental moment matching. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4652–4662, 2017.

[LKK+18] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits (JSSC)*, 54(1):173–185, 2018.

[LKS+13] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency DRAM: A low latency and low cost DRAM architecture. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 615–626, Feb 2013.

[LLL+16] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. DaDianNao: A neural network supercomputer. *IEEE Transactions on Computers (TC)*, 66(1):73–88, 2016.

[LLS$^+$20] Timothée Lesort, Vincenzo Lomonaco, Andrei Stoian, Davide Maltoni, David Filliat, and Natalia Díaz-Rodríguez. Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges. *Information Fusion*, 58:52–68, 2020.

[LM76] L. Levine and W. Meyers. Special feature: Semiconductor memory reliability with error detecting and correcting codes. *Computer*, 9(10):43–50, 1976.

[LTL$^+$19] Fang Liu, Guoming Tang, Youhuizi Li, Zhiping Cai, Xingzhou Zhang, and Tongqing Zhou. A survey on edge computing systems and tools. *Proceedings of the IEEE*, 107(8):1537–1562, 2019.

[LV62] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[LYL$^+$17] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, 2017.

[LYL$^+$18] Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 343–348, 2018.

[M$^+$18] R. Miotto et al. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246, 2018.

[Maa97] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[MAD07] Abigail Morrison, Ad Aertsen, and Markus Diesmann. Spike-timing-dependent plasticity in balanced random networks. *Neural Computation*, 19(6):1437–1467, 2007.

[MAD$^+$20] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis. Deep learning-based vehicle behavior prediction for autonomous driving applications: A review. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–15, 2020.

[MC89] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109 – 165, 1989.

[MET+17] Jamal Lottier Molin, Adebayo Eisape, Chetan Singh Thakur, Vigil Varghese, Christian Brandli, and Ralph Etienne-Cummings. Low-power, low-mismatch, highly-dense array of VLSI Mihalas-Niebur neurons. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[MGE23] Farhad Modaresi, Matthew Guthaus, and Jason K. Eshraghian. OpenSpike: An OpenRAM SNN accelerator. 2023.

[MGNDM19] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, and Timothée Masquelier. SpykeTorch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron. *Frontiers in Neuroscience (FNINS)*, 13, 2019.

[MHMS18] Alberto Marchisio, Muhammad Abdullah Hanif, Maurizio Martina, and Muhammad Shafique. Prunet: Class-blind pruning method for deep neural networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018.

[MHN13] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *30th International Conference on Machine Learning (ICML)*, volume 30, page 3, 2013.

[Mic10] Micron. Micron 2gb: x4, x8, x16 ddr3 sdram. datasheet mt41j128m16ha-12. 2010.

[Mit16] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Survey (CSUR)*, 48(4), mar 2016.

[MKK+20] Romain Mercier, Cédric Killian, Angeliki Kritikakou, Youri Helen, and Daniel Chillet. Multiple permanent faults mitigation through bit-shuffling for network-on-chip architecture. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 205–212, 2020.

[MKK+22] Romain Mercier, Cédric Killian, Angeliki Kritikakou, Youri Helen, and Daniel Chillet. BiSuT: A NoC-based bit-shuffling technique for multiple permanent faults mitigation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 41(7):2276–2289, 2022.

[ML81] Catherine Morris and Harold Lecar. Voltage oscillations in the barnacle giant muscle fiber. *Biophysical journal*, 35(1):193–213, 1981.

[MLN+12] Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. In *39th Annual International Symposium on Computer Architecture (ISCA)*, page 37–48, 2012.

[mmc] mmclassification. Model Zoo ImageNet. `https://mmclassification.readthedocs.io/en/latest/model_zoo.html`. Accessed: 2023-01-01.

[MMS$^+$11] Ammar Mohemmed, Satoshi Matsuda, Stefan Schliebs, Kshitij Dhoble, and Nikola Kasabov. Optimization of spiking neural networks with dynamic synapses for spike sequence generation using pso. In *The 2011 International Joint Conference on Neural Networks (IJCNN)*, pages 2969–2974, 2011.

[MN09] Ştefan Mihalaş and Ernst Niebur. A generalized linear integrate-and-fire neural model produces diverse spiking behaviors. *Neural Computation*, 21(3):704–718, 2009.

[MNA$^+$18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[MPC16] Homay Danaei Mehr, Huseyin Polat, and Aydin Cetin. Resident activity recognition in smart homes by using artificial neural networks. In *2016 4th International Istanbul Smart Grid Congress and Fair (ICSG)*, pages 1–5, 2016.

[MPN$^+$16] Christian Mayr, Johannes Partzsch, Marko Noack, Stefan Hänzsche, Stefan Scholze, Sebastian Höppner, Georg Ellguth, and Rene Schüffny. A biological-realtime neuromorphic system in 28 nm CMOS using low-leakage switched capacitor circuits. *IEEE Transactions on Biomedical Circuits and Systems (TBCAS)*, 10(1):243–254, 2016.

[MQSI17] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems (TBCAS)*, 12(1):106–122, 2017.

[MSMK12] Ammar Mohemmed, Stefan Schliebs, Satoshi Matsuda, and Nikola Kasabov. Span: Spike pattern association neuron for learning spatio-temporal spike patterns. *International Journal of Neural Systems*, 22(04):1250012, 2012.

[MUDV17] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI.

In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247, 2017.

[Muk11] Shubu Mukherjee. *Architecture design for soft errors.* Morgan Kaufmann, 2011.

[NFB+19] Alexander Neckar, Sam Fok, Ben V. Benjamin, Terrence C. Stewart, Nick N. Oza, Aaron R. Voelker, Chris Eliasmith, Rajit Manohar, and Kwabena Boahen. Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proceedings of the IEEE*, 107(1):144–164, 2019.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[NLSU+21] Seyed Saber Nabavi Larimi, Behzad Salami, Osman S. Unsal, Adrián Cristal Kestelman, Hamid Sarbazi-Azad, and Onur Mutlu. Understanding power consumption and reliability of high-bandwidth memory with voltage underscaling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 517–522, 2021.

[NMZ19] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine (MSP)*, 36(6):51–63, 2019.

[NPL+20] V. P. Nambiar, J. Pu, Y. K. Lee, A. Mani, T. Luo, L. Yang, E. K. Koh, M. M. Wong, F. Li, W. L. Goh, and A. T. Do. 0.5v 4.8 pj/sop 0.93$\mu$W leakage/core neuromorphic processor with asynchronous NoC and reconfigurable LIF neuron. In *2020 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 1–4, 2020.

[Nvia] Nvidia. Nvidia GeForce GTX 1060. `https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060`. Accessed: 2020-06-01.

[Nvib] Nvidia. Nvidia GeForce GTX 1080 Ti. `https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1080-ti`. Accessed: 2020-06-01.

[Nvic] Nvidia. Nvidia GeForce GTX 2080 Ti. `https://www.nvidia.com/de-at/geforce/graphics-cards/rtx-2080-ti`. Accessed: 2020-06-01.

[Nvid] Nvidia. Nvidia Jetson Nano. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`. Accessed: 2020-06-01.

[Nvie] Nvidia. Nvidia Jetson TX2. `https://developer.nvidia.com/embedded/jetson-tx2`. Accessed: 2020-06-01.

[Ope] OpenVINO. Model Zoo. `https://docs.openvino.ai/latest/omz_models_group_public.html`. Accessed: 2023-01-01.

[pap] paperswithcode. Image classification. `https://paperswithcode.com/sota/image-classification-on-imagenet`. Accessed: 2023-01-01.

[PAR$^+$21] Bharath Srinivas Prabakaran, Asima Akhtar, Semeen Rehman, Osman Hasan, and Muhammad Shafique. BioNetExplorer: Architecture-space exploration of biosignal processing deep neural networks for wearables. *IEEE Internet of Things Journal (JIOT)*, 8(17):13251–13265, 2021.

[PARR18] Priyadarshini Panda, Jason M. Allred, S. Ramanathan, and K. Roy. Asp: Learning to forget with adaptive synaptic plasticity in spiking neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):51–64, March 2018.

[PDS$^+$19] Jing Pei, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, et al. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature*, 572(7767):106–111, 2019.

[PHS20] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. DRMap: A generic DRAM data mapping policy for energy-efficient processing of convolutional neural networks. In *The 57th ACM/IEEE Design Automation Conference (**DAC**)*, pages 1–6, 2020.

[PHS21a] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. ReSpawn: Energy-efficient fault-tolerance for spiking neural networks considering unreliable memories. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[PHS21b] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. ROMANet: Fine-grained reuse-driven off-chip memory access management and data organization for deep neural network accelerators. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, pages 1–14, 2021.

[PHS21c] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. SparkXD: A framework for resilient and energy-efficient spiking neural network inference using approximate DRAM. In *The 58th ACM/IEEE Design Automation Conference (**DAC**)*, pages 1–6, 2021.

[PHS22a] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. EnforceSNN: Enabling resilient and energy-efficient spiking neural network inference considering approximate DRAMs for embedded systems. *Frontiers in Neuroscience (FNINS)*, 16, 2022.

[PHS22b] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. SoftSNN: Low-cost fault tolerance for spiking neural network accelerators under soft errors. In *59th ACM/IEEE Design Automation Conference (DAC)*, page 151–156, 2022.

[PHS23] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. RescueSNN: Enabling reliable executions on spiking neural network accelerators under permanent faults. *Frontiers in Neuroscience (FNINS)*, 17, 2023.

[PHY+14] Jongkil Park, Sohmyung Ha, Theodore Yu, Emre Neftci, and Gert Cauwenberghs. A 65k-neuron 73-mevents/s 22-pj/event asynchronous micro-pipelined integrate-and-fire array transceiver. In *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) proceedings*, pages 675–678. IEEE, 2014.

[PKCY19] Seongsik Park, Seijoon Kim, Hyeokjun Choe, and Sungroh Yoon. Fast and efficient information transmission with burst spikes in deep spiking neural networks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[PKNY20] Seongsik Park, Seijoon Kim, Byunggook Na, and Sungroh Yoon. T2FSNN: Deep spiking neural networks with time-to-first-spike coding. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[PKP+19] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54 – 71, 2019.

[PLJ20] Jeongwoo Park, Juyun Lee, and Dongsuk Jeon. A 65-nm neuromorphic image classification processor with energy-efficient training through direct spike-only feedback. *IEEE Journal of Solid-State Circuits (JSSC)*, 55(1):108–119, 2020.

[PP18] Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: Opportunities and challenges. *Frontiers in Neuroscience (FNINS)*, 12, 2018.

[PPG+13] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown, and

Steve B Furber. SpiNNaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits (JSCC)*, 48(8):1943–1953, 2013.

[PRM+17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, June 2017.

[PS20] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. FSpiNN: An optimization framework for memory-efficient and energy-efficient spiking neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):3601–3613, 2020.

[PS21a] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. Q-SpiNN: A framework for quantizing spiking neural networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[PS21b] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. SpikeDyn: A framework for energy-efficient spiking neural networks with continual and unsupervised learning capabilities in dynamic environments. In *The 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2021.

[PS22a] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. lpSpikeCon: Enabling low-precision spiking neural network processing for efficient unsupervised continual learning on autonomous agents. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[PS22b] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. tinySNN: Towards memory-and energy-efficient spiking neural networks. *arXiv preprint arXiv:2206.08656*, 2022.

[PS23a] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. Mantis: Enabling energy-efficient autonomous mobile agents with spiking neural networks. In *2023 International Conference On Automation, Robotics and Applications (ICARA)*, pages 1–5, 2023.

[PS23b] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. TopSpark: A timestep optimization methodology for energy-efficient spiking neural networks on autonomous mobile agents. *arXiv preprint arXiv:2303.01826*, 2023.

[QMC+15] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in Neuroscience (FNINS)*, 9:141, 2015.

[QWY$^+$16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded FPGA platform for convolutional neural network. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 26–35, 2016.

[RAIAS$^+$14] Mostafa Rahimi Azghadi, Nicolangelo Iannella, Said F. Al-Sarawi, Giacomo Indiveri, and Derek Abbott. Spike-based synaptic plasticity in silicon: Design, implementation, application, and challenges. *Proceedings of the IEEE (JPROC)*, 102(5):717–737, 2014.

[RBKS17] Rengarajan Ragavan, Benjamin Barrois, Cedric Killian, and Olivier Sentieys. Pushing the limits of voltage over-scaling for error-resilient applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 476–481, 2017.

[RCK$^+$23] Nitin Rathi, Indranil Chakraborty, Adarsh Kosta, Abhronil Sengupta, Aayush Ankit, Priyadarshini Panda, and Kaushik Roy. Exploring neuromorphic computing based on spiking neural networks: Algorithms to hardware. *ACM Computing Survey*, 55(12), March 2023.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016.

[REHS$^+$16] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, Jorg Henkel, and Jörg Henkel. Architectural-space exploration of approximate multipliers. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[RFZJ13] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys*, 46(1), Jul 2013.

[RH89] R. M. Rose and J. L. Hindmarsh. The assembly of ionic currents in a thalamic neuron i. the three-dimensional model. *Proceedings of the Royal Society of London. B. Biological Sciences*, 237(1288):267–288, 1989.

[RKL$^+$19] Bodo Rückauer, Nicolas Känzig, Shih-Chii Liu, Tobi Delbruck, and Yulia Sandamirskaya. Closing the accuracy gap in an event-based visual recognition task. *arXiv preprint arXiv:1906.08859*, 2019.

[RLH$^+$17] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience (FNINS)*, 11, 2017.

[RLIS21] Mehul Rastogi, Sen Lu, Nafiul Islam, and Abhronil Sengupta. On the self-repair role of astrocytes in STDP enabled unsupervised SNNs. *Frontiers in Neuroscience (FNINS)*, 14:603796, 2021.

[ROD$^+$10] Alexander Russell, Garrick Orchard, Yi Dong, Ştefan Mihalaş, Ernst Niebur, Jonathan Tapson, and Ralph Etienne-Cummings. Optimization methods for spiking neurons and networks. *IEEE Transactions on Neural Networks*, 21(12):1950–1962, 2010.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[RPR19] Nithin Rathi, Priyadarshini Panda, and Kaushik Roy. Stdp-based pruning of connections and weight quantization in spiking neural networks for energy-efficient recognition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(4):668–677, April 2019.

[RS97] Berthold Ruf and Michael Schmitt. Hebbian learning in networks of spiking neurons using temporal coding. In *International Work-Conference on Artificial and Natural Neural Networks (IWANN)*, pages 380–389, 1997.

[RSPR20] Nitin Rathi, Gopalakrishnan Srinivasan, Priyadarshini Panda, and Kaushik Roy. Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[RTGM13] Bharathwaj Raghunathan, Yatish Turakhia, Siddharth Garg, and Diana Marculescu. Cherry-picking: Exploiting process variations in dark-silicon homogeneous chip multi-processors. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 39–44, 2013.

[RVG$^+$17] Arnab Roy, Swagath Venkataramani, Neel Gala, Sanchari Sen, Kamakoti Veezhinathan, and Anand Raghunathan. A programmable event-driven architecture for evaluating spiking neural networks. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, July 2017.

[S$^+$17] G. Srinivasan et al. Spike timing dependent plasticity based enhanced self-learning for efficient pattern recognition in spiking neural networks. In *Proc. of the Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1847–1854, May 2017.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[SBG+10] Johannes Schemmel, Daniel Brüderle, Andreas Grübl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950. IEEE, 2010.

[SBL+11] Jae-sun Seo, Bernard Brezzo, Yong Liu, Benjamin D. Parker, Steven K. Esser, Robert K. Montoye, Bipin Rajendran, José A. Tierno, Leland Chang, Dharmendra S. Modha, and Daniel J. Friedman. A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, 2011.

[SCB19] Arthur Stoutchinin, Francesco Conti, and Luca Benini. Optimally scheduling CNN convolutions for efficient memory access. *arXiv preprint arXiv:1902.01492*, 2019.

[SCSR00] Gregory D. Smith, Charles L. Cox, S. Murray Sherman, and John Rinzel. Fourier analysis of sinusoidally driven thalamocortical relay neurons and a minimal integrate-and-fire-or-burst model. *Journal of Neurophysiology*, 83(1):588–610, 2000.

[SCYE17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SESA+21] Theofilos Spyrou, Sarah A El-Sayed, Engin Afacan, Luis A Camuñas-Mesa, Bernabé Linares-Barranco, and Haralampos-G. Stratigopoulos. Neuron fault tolerance in spiking neural networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, February 2021.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3645–3650, 2019.

[SGM20] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. *AAAI Conference on Artificial Intelligence (AAAI*, 34(09):13693–13696, Apr. 2020.

[SH23] Ayesha Siddique and Khaza Anuarul Hoque. Improving reliability of spiking neural networks through fault aware threshold voltage optimization. *arXiv preprint arXiv:2301.05266*, 2023.

[SJ20] Clemens JS Schaefer and Siddharth Joshi. Quantizing spiking neural networks with integers. In *International Conference on Neuromorphic Systems (ICONS)*, 2020.

[SKR18] Muhammad Shafique, Faiq Khalid, and Semeen Rehman. Intelligent security measures for smart cyber physical systems. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 280–287, 2018.

[SLBS20] Martino Sorbaro, Qian Liu, Massimo Bortone, and Sadique Sheik. Optimizing the energy consumption of spiking neural networks for neuromorphic applications. *Frontiers in Neuroscience (FNINS)*, 14, 2020.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[SLL+18] Dongjoo Shin, Jinmook Lee, Jinsu Lee, Juhyoung Lee, and Hoi-Jun Yoo. DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture. *IEEE Micro*, 38(5):85–93, 2018.

[SLM+19a] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 304–317, 2019.

[SLM19b] Xueyuan She, Yun Long, and Saibal Mukhopadhyay. Fast and low-precision learning in GPU-accelerated spiking neural network. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 450–455, 2019.

[SLS23] Fei Su, Chunsheng Liu, and Haralampos-G. Stratigopoulos. Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives. *IEEE Design & Test*, pages 1–1, 2023.

[SMWPH21] Muhammad Shafique, Alberto Marchisio, Rachmad Vidya Wicaksana Putra, and Muhammad Abdullah Hanif. Towards energy-efficient and secure edge AI: A cross-layer framework ICCAD special session paper. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[SNFK20] Aseem Sayal, S. S. Teja Nibhanupudi, Shirin Fathima, and Jaydeep P. Kulkarni. A 12.08-tops/w all-digital time-domain CNN engine using bi-directional memory delay lines for energy efficient edge computing. *IEEE Journal of Solid-State Circuits (JSSC)*, 55(1):60–75, 2020.

[SNT$^+$20a] M. Shafique, M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi. Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead. *IEEE Design Test*, 37(2):30–57, 2020.

[SNT$^+$20b] Muhammad Shafique, Mahum Naseer, Theocharis Theocharides, Christos Kyrkou, Onur Mutlu, Lois Orosa, and Jungwook Choi. Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead. *IEEE Design & Test*, 37(2):30–57, 2020.

[SO18] Sumit Bam Shrestha and Garrick Orchard. SLAYER: Spike layer error reassignment in time. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31. Curran Associates, Inc., 2018.

[SPH$^+$19] Daniel J. Saunders, Devdhar Patel, Hananel Hazan, Hava T. Siegelmann, and Robert Kozma. Locally connected spiking neural networks for unsupervised feature learning. *Neural Networks*, 119:332 – 340, 2019.

[SPMJ$^+$20] Catherine D. Schuman, J. Parker Mitchell, J. Travis Johnston, Maryam Parsa, Bill Kay, Prasanna Date, and Robert M. Patton. Resilience and robustness of spiking neural networks for neuromorphic systems. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2020.

[SPP$^+$17] Catherine D Schuman, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

[SPS$^+$18] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[SSKR18] Daniel J. Saunders, Hava T. Siegelmann, Robert Kozma, and Mikl´os Ruszink´o. STDP learning of image patches with convolutional spiking neural networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2018.

[SSL11] Miloš Stanisavljević, Alexandre Schmid, and Yusuf Leblebici. Reliability, faults, and fault tolerance. In *Reliability of Nanoscale Circuits and Systems*, pages 7–18. Springer, 2011.

[SSUCK18] Behzad Salami, Osman S. Unsal, and Adrian Cristal Kestelman. Comprehensive evaluation of supply voltage underscaling in FPGA on-chip memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 724–736, 2018.

[SSYC21] Jan Stuijt, Manolis Sifalakis, Amirreza Yousefzadeh, and Federico Corradi. $\mu$Brain: An event-driven and fully synthesizable architecture for spiking neural networks. *Frontiers in Neuroscience (FNINS)*, page 538, 2021.

[Ste65] Richard B. Stein. A theoretical analysis of neuronal variability. *Biophysical Journal*, 5(2):173–194, 1965.

[SVR17] Sanchari Sen, Swagath Venkataramani, and Anand Raghunathan. Approximate computing for spiking neural networks. In *Design, Automation Test in Europe Conf. Exhibition (DATE), 2017*, pages 193–198, March 2017.

[SVVC07] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: Theory, applications and implementations. In *15th European Symposium on Artificial Neural Networks (ESANN)*, pages 471–482, 2007.

[SZ98] Charles F Stevens and Anthony M Zador. Novel integrate-and-re-like model of repetitive firing in cortical neurons. *American Physiological Society*, 1998.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[Sze00] Hung Y. Sze. Circuit and method for rapid checking of error correction codes using cyclic redundancy check, July 18 2000. US Patent 6,092,231.

[TG98] Simon Thorpe and Jacques Gautrais. Rank order coding. *Computational Neuroscience: Trends in Research, 1998*, pages 113–118, 1998.

[TGK+19] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

[THG17] Cesar Torres-Huitzil and Bernard Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017.

[TKP20] Saurabh Tewari, Anshul Kumar, and Kolin Paul. Bus width aware off-chip memory access minimization for CNN accelerators. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 240–245, 2020.

[TYO+17] Fengbin Tu, Shouyi Yin, Peng Ouyang, Shibin Tang, Leibo Liu, and Shaojun Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 25(8):2220–2233, 2017.

[UAH+18] Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Junichiro Kadomoto, Tomoki Miyata, Mototsugu Hamada, Tadahiro Kuroda, and Masato Motomura. QUEST: A 7.49tops multi-purpose log-quantized DNN inference engine stacked on 96mb 3D SRAM using inductive-coupling technology in 40nm CMOS. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 216–218, 2018.

[Vai] Lionel Sujay Vailshery. Number of internet of things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`. Accessed: 2023-01-01.

[vBKM+22] Mart van Baalen, Brian Kahne, Eric Mahurin, Andrey Kuzmin, Andrii Skliar, Markus Nagel, and Tijmen Blankevoort. Simulated quantization, real power savings. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2757–2761, 2022.

[VCC+13] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2013.

[VCRR15] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[VDNA19] Elena-Ioana Vatajelu, Giorgio Di Natale, and Lorena Anghel. Special session: Reliability of hardware-implemented spiking neural networks (snn). In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–8, 2019.

[vdVST20] Gido M van de Ven, Hava T Siegelmann, and Andreas S Tolias. Brain-inspired replay for continual learning with artificial neural networks. *Nature communications*, 11(1):1–14, 2020.

[VMA+20] Valerio Venceslai, Alberto Marchisio, Ihsen Alouani, Maurizio Martina, and Muhammad Shafique. NeuroAttack: Undermining spiking neural networks security through externally triggered bit-flips. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

[VZBT10] Ramakrishna Vadlamani, Jia Zhao, Wayne Burleson, and Russell Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 27–32, 2010.

[Wan19] Yi Wang. Efficient spiking neural network training and inference with reduced precision memory and computing. *IET Computers & Digital Techniques*, 13:397–404(7), September 2019.

[WBK06] Simei Gomes Wysoski, Lubica Benuskova, and Nikola Kasabov. On-line learning with structural adaptation in a network of spiking neurons for visual pattern recognition. In *Proceedings of International Conference on Artificial Neural Networks (ICANN)*, pages 61–70, 2006.

[WDL+18] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience (FNINS)*, 12, 2018.

[WEH+19] Junshi Wang, Masoumeh Ebrahimi, Letian Huang, Xuan Xie, Qiang Li, Guangjun Li, and Axel Jantsch. Efficient design-for-test approach for networks-on-chip. *IEEE Transactions on Computers (TC)*, 68(2):198–213, 2019.

[Wil99] Hugh R. Wilson. Simplified dynamics of human and mammalian neocortical neurons. *Journal of Theoretical Biology*, 200(4):375–388, 1999.

[WKE+20] Weier Wan, Rajkumar Kubendran, S. Burc Eryilmaz, Wenqiang Zhang, Yan Liao, Dabin Wu, Stephen Deiss, Bin Gao, Priyanka Raina, Siddharth Joshi, Huaqiang Wu, Gert Cauwenberghs, and H.-S. Philip Wong. 33.1 a 74 TMACS/W CMOS-RRAM neurosynaptic core with dynamically reconfigurable dataflow and in-situ transposable weights for probabilistic graphical models. In *2020 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 498–500, 2020.

[WL91] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2(4):452–471, 1991.

[WNL16] Sebastian Werner, Javier Navaridas, and Mikel Luján. A survey on design approaches to circumvent permanent faults in networks-on-chip. *ACM Computing Surveys*, 48(4), March 2016.

[XMK16] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test (DnT)*, 33(1):8–22, 2016.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[XTAQ20] Qian Xu, Md Tanvir Arafin, and Gang Qu. MIDAS: Model inversion defenses using an approximate memory system. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–4, 2020.

[XZLZ20] Hongwei Xu, Jie Zhang, Youlong Lv, and Peng Zheng. Hybrid feature selection for wafer acceptance test parameters in semiconductor manufacturing. *IEEE Access*, 8:17320–17330, 2020.

[YAA$^+$23] Jason Yik, Soikat Hasan Ahmed, Zergham Ahmed, Brian Anderson, Andreas G Andreou, Chiara Bartolozzi, Arindam Basu, Douwe den Blanken, Petrut Bogdan, Sander Bohte, et al. NeuroBench: Advancing neuromorphic computing through collaborative, fair and representative benchmarking. *arXiv preprint arXiv:2304.04640*, 2023.

[YLH$^+$18] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE Access*, 6:6900–6919, 2018.

[YLP$^+$17] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[YSE$^+$22] İsmail Emir Yüksel, Behzad Salami, Oğuz Ergin, Osman Sabri Unsal, and Adrián Cristal Kestelman. MoRS: An approximate fault modeling framework for reduced-voltage SRAMs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 41(6):1663–1673, 2022.

[YYY$^+$18] Zhe Yuan, Jinshan Yue, Huanrui Yang, Zhibo Wang, Jinyang Li, Yixiong Yang, Qingwei Guo, Xueqing Li, Meng-Fan Chang, Huazhong Yang, and Yongpan Liu. Sticker: A 0.41-62.1 tops/w 8bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers. In *IEEE Symposium on VLSI Circuits (VLSIC)*, pages 33–34, 2018.

[ZAJ$^+$19] Maheen Zahid, Fahad Ahmed, Nadeem Javaid, Raza Abid Abbasi, Hafiza Syeda Zainab Kazmi, Atia Javaid, Muhammad Bilal, Mariam Akbar, and Manzoor Ilahi. Electricity price and load forecasting using enhanced convolutional neural network and enhanced support vector regression in smart grids. *Electronics*, 8(2), 2019.

[ZCG$^+$20] Chenglong Zou, Xiaoxin Cui, Jiexian Ge, Hanghang Ma, and Xinan Wang. A novel conversion method for spiking neural network using median quantization. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[ZDZ+16] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[ZGBG18] Jeff Jun Zhang, Tianyu Gu, Kanad Basu, and Siddharth Garg. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6, 2018.

[ZLK+19] Jeff Jun Zhang, Kang Liu, Faiq Khalid, Muhammad Abdullah Hanif, Semeen Rehman, Theocharis Theocharides, Alessandro Artussi, Muhammad Shafique, and Siddharth Garg. Building robust machine learning systems: Current progress, research challenges, and opportunities. In *56th Annual Design Automation Conference (DAC)*, pages 1–4, 2019.

[ZLL+21] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W. Keckler, and Zhengya Zhang. SNAP: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference. *IEEE Journal of Solid-State Circuits (JSSC)*, 56(2):636–647, 2021.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[ZSF+19] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(11):2072–2085, Nov 2019.