



UNIVERSITÀ
DI TRENTO

Department of
Information Engineering and Computer Science

Doctoral Programme in
Information Engineering and Computer Science

EFFICIENT PROCESSING OF SPIKING NEURAL NETWORKS: A MEMORY-BASED APPROACH

Muath Abu Lebdeh

Advisor

Kasim Sinan Yildirim
University of Trento

Co-Advisor

Davide Brunelli
University of Trento

October 2025

Acknowledgment

This work would not have been possible without the support of the people around me. I am grateful to my parents for their continuous encouragement to complete my PhD. I would also like to sincerely thank my advisors, Kasim Sinan Yildirim and Davide Brunelli, for their guidance and support. Special thanks to Elisabetta Chicca for giving me the opportunity to visit her research group in Groningen and for the enriching my learning experience. Thanks to Edoardo Patti and Francesco Conti for their valuable feedback to improve this dissertation. Finally, I extend my gratitude to the University of Trento for the support and resources provided throughout my PhD journey.

Abstract

Neuromorphic systems have been developed with the aim of mimicking biological systems in terms of functionality and processing efficiency. Spiking neural networks (SNNs) are widely used as the computing model for the neuromorphic system. The neurons in the SNN communicate using spikes, which allows the SNN to be efficiently implemented on hardware. In addition, the SNNs are based on the spiking neuron, a dynamical neuron that extracts temporal features at the neuron level.

On the other hand, the dynamicity of the spiking neuron is implemented using one or more state variables, which introduces a serious memory overhead in terms of memory operations (read/write), runtime memory requirement, and the utilization of hierarchical hardware memory. This memory overhead degrades the SNNs' processing efficiency and increases their hardware cost. In other words, the memory overhead at the neuron level offsets the computational benefits gained from the spiking communication between neurons at the network level. This memory overhead problem has not been deeply addressed in the field of neuromorphic computing, as the SNNs are still not at large scale compared to the developed neuromorphic hardware.

This dissertation aims to address the memory overhead in SNNs to increase their processing efficiency and decrease their hardware cost. The adopted research method focuses mainly on the algorithmic level, that is, on the design of neural network architectures. The design of the neural network architecture is constrained by two main metrics: (1) maintaining the accuracy of the AI task while (2) optimizing their memory efficiency when running on a hardware platform.

All proposed solutions narrow the dataflow of the SNN's computation, which reduces the memory overhead. The research carried out in this dissertation has resulted in four main contributions. The first two contributions are based on the idea of specialization, which is an adopted concept from conventional artificial neural networks (ANNs). Specialization helps to reduce memory overhead when deployed in SNNs. The next two contributions utilize the inherent dynamicity of the spiking neuron to reduce the runtime memory requirement and improve the utilization of hierarchical hardware memory.

The proposed solutions allow the SNNs to be more efficient in terms of memory, and hence in terms of the processing efficiency and cost. Task specialization improves the energy and latency of the SNNs inference by at least 10x. Per layer specialization eliminates the training burden of the task specialization approach, while improving the processing efficiency (in terms of latency) by up to 3x. The temporal dynamics of the spiking neuron were analyzed in terms of feature extraction, especially for images. The

temporal dynamics of biologically-plausible spiking neurons were exploited to process a whole image using a single spiking neuron. This approach allows for reducing the memory requirement of a convolutional SNN by at least 4x, and optimizes the processing efficiency of the SNN based on the available hardware memory.

The approaches and network architectures proposed in this dissertation emphasize the importance of memory in the design of SNNs, increases the processing efficiency of the SNNs, reduces the hardware cost that implements the SNNs, and opens the ground for a new set of application-specific solutions that exploit the temporal dynamics of the spiking neuron.

Contents

Acknowledgment	i
Abstract	iii
List of Figures	vii
List of Tables	xii
1 Introduction	1
1.1 Neuromorphics: Efficient AI systems	1
1.2 Research Problem	4
1.3 Research Objective and Methodology	7
1.4 Dissertation Contribution and Outline	8
2 Background	11
2.1 Spiking Neuron Networks	11
2.2 Memory Overhead of SNNs	14
2.3 Existing Solutions to Alleviate the Memory Overhead Caused by the SNN's Dynamicity	18
3 Task Specialization for Efficient Processing of Spiking Neural Networks	23
3.1 Introduction	23
3.2 Proposed Solution: Specialized Spiking Neural Networks	27
3.2.1 Reducing the width of the SNN model	27
3.2.2 Designing the Specialized Tasks	29
3.2.3 Training via Heterogeneous Batching	29
3.2.4 Sub-SNN Structuring	32
3.2.5 Introducing Dependencies Between the Deepest Features Via Simple Aggregation Function	33
3.2.6 The Efficiency Potential of Using Sub-SNNs	34
3.3 Experiments	35
3.3.1 Evaluation Methodology	35
3.3.2 Results	37
3.4 Limitations	42

3.5	Conclusion	44
4	Improving the Specialization by Per-Layer Specialization	45
4.1	Introduction	45
4.2	Proposed Solution: Per Layer Specialization	47
4.2.1	Preliminary Observations: Unique Representations of Semanti- cally Different Classes at Deeper Layers	47
4.2.2	Gated Specialized Models	49
4.2.3	Balanced Usage of Experts	50
4.3	Experiments and Discussion	52
4.3.1	Evaluation Methods	52
4.3.2	Results and Discussions	53
4.4	Conclusion	61
5	Utilizing Spiking Neurons of Higher Dynamical Complexity	63
5.1	Introduction	63
5.2	Proposed Solution: Efficient Processing Utilizing the Temporal Dynam- ics of the Spiking Neuron	64
5.2.1	Challenge of Extracting the Global Context Utilizing the Tempo- ral Dynamics of the Spiking Neuron	64
5.2.2	Temporal Dynamics of LIF Spiking Neuron	65
5.2.3	Higher Complexity Spiking Neuron for Global Context Extraction	66
5.2.4	Sequential Processing of Images using AdExLIF	69
5.3	Experiments	71
5.3.1	Transient Response	72
5.3.2	Frequency Response	74
5.3.3	Sequential Convolution of Images	76
5.4	Conclusion	80
6	Spatiotemporal Convolutional Spiking Neural Networks for Memory- Efficient Processing	81
6.1	Introduction	81
6.2	Proposed Solution: Spatiotemporal Convolutional SNN	82
6.2.1	Proposed Network Architecture	82
6.2.2	Network of AdExLIF	86
6.3	Experiments	87
6.3.1	Accuracy and Theoretical Efficiency	87
6.3.2	Maximum Runtime Memory	90
6.3.3	Latency and Throughput	92
6.4	Limitations	96
6.5	Conclusion	97
7	Conclusion	99
7.1	Future Work	100
7.2	List of Publications	102

Bibliography	103
---------------------	------------

List of Figures

1.1	The ideal neuromorphic system is engineered across all computing levels	2
1.2	Examples of neuromorphic designs across the different abstraction levels of the design stack. This dissertation focuses on neural network designs that are oriented for efficient memory usage.	3
1.3	Overview of the Contributions	8
2.1	The spiking neuron receives spikes as input and produces spikes as output. The input spikes are accumulated in the membrane potential ($v^L(t)$), and an output spike is generated if $v^L(t) > v_{th}$	12
2.2	Maximum runtime memory for different numbers of layers	16
2.3	Comparison between ANN and SNN in the the utilization of a hierarchical memory system. As both networks are performed layer by layer, an increased batch size pushes the memory system closer to its communication bottleneck; i.e., intermediate results in every layer does not fit the intermediate memories (caches).	17
3.1	A baseline SNN is replaced by multiple independent SNNs of narrow widths. The depth of each narrow SNNs is the same as the baseline SNN. The sum of the features in all narrow SNN at a specific layer equals the number of features at that layer in the baseline SNN. This property allows both structures to have similar memorization and generalization capacities.	27
3.2	The SNN inference is performed as a collaborative effort of multiple specialized SNNs. These SNNs can be mapped to local computing resources, such as the multiple cores presented in this figure. Specialized and small SNNs can be executed sequentially or in parallel.	28
3.3	Training each sub-SNN with heterogeneous sets of batches. Set 1 contains 50% of the samples from the target object and the other 50% from all alien objects. The samples in set 2 are sampled with equal percentages from all objects. Set 3 is the same as Set 1 except that this set contains larger batches.	31
3.4	A sub-SNN is structured by reducing the width of the baseline SNN, then training it on a subset of the dataset, then comparing the sub-SNN accuracy to the baseline SNN. This process is repeated until the scaled sub-SNN does not meet the accuracy of the baseline SNN.	32

3.5	This figure shows the spiking intensity of different sub-SNNs (in this example we have 10) when samples from different objects is applied (here we have 10 objects)	33
3.6	Abstraction of different typical computing architectures.	35
3.7	Spikes per step: (a) MNIST-MLP, (c) CIFAR10-VGG9, (e) CIFAR10-RESNET, (g) KWS-3FCLs. Normalized spikes per step: (b) MNIST-MLP, (d) CIFAR10-VGG9, (f) CIFAR10-RESNET, (h) KWS-3FCLs.	39
3.8	Energy: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.	40
3.9	Latency: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.	41
4.1	Projection of features at different layers into 2D plane for CustomResNet-9 trained on CIFAR10 dataset. (a) shows the projections of features for three semantically different classes: ‘airplane’ (magenta), ‘automobile’ (blue), and ‘cat’ (yellow). (b) shows the projections of features for 2 semantically similar classes, ‘cat’ (yellow) and ‘dog’ (green), and one semantically different class, ‘airplane’ (magenta). The depth of the layer increases from left to right (i.e., the rightmost figure shows the projection of the deepest features).	48
4.2	The proposed architecture for task specialization per layer. The gating function routes the input features to only one of the specialized experts.	49
4.3	The gated layer used during training. One expert is selected at every time step.	50
4.4	Inference latency per sample with different input batch sizes. (a) Shows the latency for architectures trained on CIFAR10 dataset, while (b) shows the latency for architectures trained on fashion-mnist dataset.	56
4.5	The membrane potentials memory required in every time step.	58
4.6	The activated experts in every layer for samples from different classes in CustomResNet-9 architecture trained on CIFAR10.	58
4.7	The activated experts in every step for samples from different classes in CustomResNet-9 trained in CIFAR10. The results are shown for (a) layer 4, (b) layer 6 and (c) layer 8.	59
5.1	The theoretical linear frequency response of the LIF spiking neuron; the magnitude of the linear frequency response of the neuron.	67
5.2	The theoretical linear frequency response of the AdExLIF spiking neuron. g_L was fixed to 0.8 and a is tuned from 0 to 0.9. This shows that AdExLIF can be tune to band widths of temporal features.	68
5.3	The theoretical linear frequency response of the AdExLIF spiking neuron. a was fixed to 0.8 and g_L is tuned from 0 to 0.9. This shows that AdExLIF can be tune to detect different ranges of temporal features.	68
5.4	The theoretical linear frequency response of the AdExLIF spiking neuron. a was fixed to 0.99 and g_L is tuned from 0 to 0.9. This shows that AdExLIF can be tune to detect narrow range of temporal features.	69

5.5	Different impulse responses (membrane potential response to an impulse) of the AdExLIF spiking neuron.	70
5.6	Transient response of an AdExLIF neuron parameterized with $g_L=0.5$, $a=0.99$ and $u_{th}=30$. The neuron resets the membrane potential and adaptation variables to zero when outputting a spike.	73
5.7	Transient response of the LIF neuron parameterized with $g_L=0.5$ and $u_{th}=0.8$. The neuron resets the membrane potential and adaptation variables to zero when spike.	74
5.8	Input frequency to output frequency mapping of AdExLIF (parameters: $a=0.8$, $g_L=0.8$, $u_{th}=2$). This shows the transient response of an input sinusoid that persists for long period of time. The neuron spiking output is also periodic, where an a periodic sinusoidal signal is transformed to a periodic spiking signal.	75
5.9	experimental frequency response of a frequency decomposer example; AdExLIF: $a=0.99$, $u_{th}=30$, $V_T=100$	76
5.10	experimental frequency response of a frequency decomposer example; 3 decomposers; cosine mixer (input frequency shifter); AdExLIF: $a=0.99$, $v_{th}=30$, $V_T=100$	77
5.11	impulse responses of 10 different neurons, parameterized with fixed a ($=0.9$) and varying g_L from 0 to 0.9	77
5.12	Image processed sequentially (pixel by pixels) by the neurons with impulse responses in previous Fig. 5.11 (location of the output image is filtering with the neuron with the corresponding location in Fig. 5.11).	78
5.13	Transeint response: AdExLIF neuron, $g_L=0.5$, $a=0.99$, $u_{th}=30$, reset to zero when spike	80
6.1	An example of patching an MNIST image. This 28x28 image is patched into 4x4 (16) patches, each of 7x7 (49) pixels. The local feature in every patch is extracted spatially, while the global features between different patches are extracted temporally.	84
6.2	The proposed spatiotemporal convolutional SNN (CONV-SNN) architecture processes an input image as a sequence of 2D patches (tokens). This reduces the memory required for the intermediate results. This architecture extract the global visual context temporally, utilizing the intrinsic temporal dynamics of the spiking neuron.	85
6.3	The spatiotemporal spiking convolutional layer consists of two types of convolutions. The FIR spatial convolution is used to extract the local context from every 2D patch. The IIR temporal convolution is used to extract the global context from the spatially processed sequence of 2D patches.	85
6.4	batch size of 256; GPU Memory (using <code>torch.cuda.max_memory_allocated</code>)	90
6.5	Latency for different batch sizes (to emulate different memory resources). This was measured for 3 layers networks processed by a CPU.	92

6.6	Throughput (inferences per second) for different batch sizes. This was measured for 3 layers networks processed by a CPU.	93
6.7	Latency for different batch sizes. This was measured for 3 layers networks processed by a GPU.	95
6.8	Throughput (inferences per second) for different batch sizes. This was measured for 3 layers networks processed by a GPU.	96

List of Tables

2.1	Comparison of the solutions in the literature that aims at alleviating the memory overhead due to the dynamicity of the SNNs. These solutions are classified under 5 classes: Quantization, pruning, reducing activity, reducing the number of steps, and membrane potential sharing.	19
3.1	Evaluated baseline SNN architectures	36
3.2	Accuracy (%) on different networks and datasets	38
4.1	Accuracy comparison on different architectures and datasets	53
6.1	Performance comparison of different models.	87
6.2	Hardware specifications	91

Chapter 1

Introduction

1.1 Neuromorphics: Efficient AI systems

Artificial neural networks (ANNs) have been successful in achieving outperforming accuracy for complex intelligence tasks, such as computer vision [108, 45], speech recognition [68, 86], machine translation [16, 15], general-purpose AI [119, 28, 127, 44]. On the other hand, this outperforming accuracy is achieved at the expense of high computational and memory requirements [106, 107, 18]. The huge computational and memory requirements prevent large-scale ANNs from being deployed in modern applications that rely on resource-constrained computing systems. One extreme example is to deploy a deep ANN on a batteryless microcontroller with a response in real-time [31]. This example shows a sharp contrast between the requirements of a deep ANN and the constraints from the host computing system and the emerging applications.

Neuromorphic computing aims to improve the processing efficiency of AI systems by proposing brain-inspired solutions across the computer design stack, ranging from the neural network [95, 50], to the neuron model neuron [41, 11, 77], to the hardware architecture [21, 81, 32, 33], and even to the circuit and device levels [71, 40, 74, 69] (as in Fig. 1.1). The motivation behind this idea is that the brain has the property of solving intelligence tasks with high accuracy while consuming a low level of energy and power.

To gain insight into how brain-inspired designs can be computationally efficient, let us look at the spiking neural networks (SNNs); the computational model of the neuromorphic system. SNNs are neural networks that use a biologically inspired neuron known as the spiking neuron. The spiking neuron quantizes its output into either ‘1’ (spike) or ‘0’ (no spike): (1) This improves the communication overhead as neurons

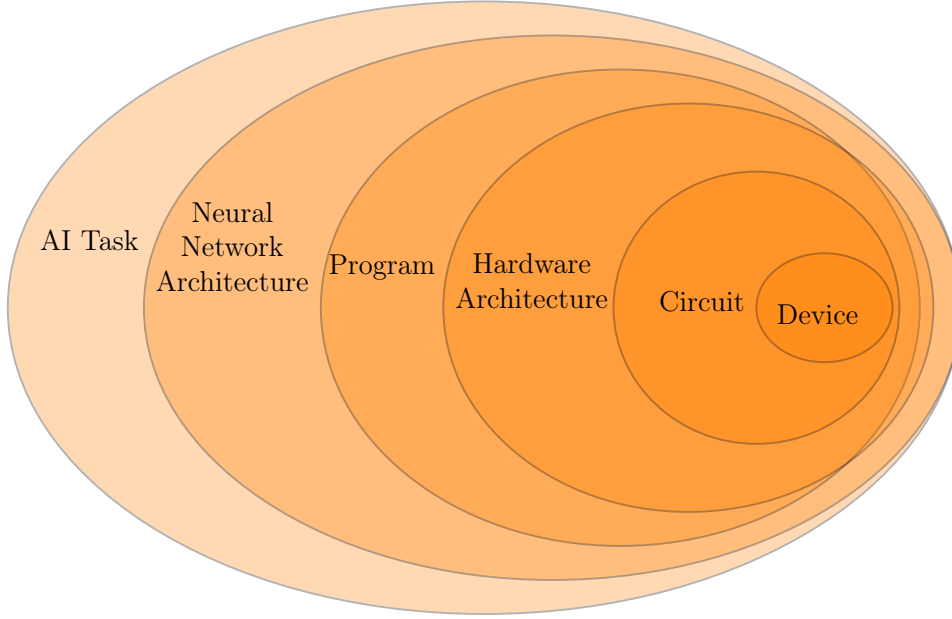


Figure 1.1: The ideal neuromorphic system is engineered across all computing levels

only need a single bit to communicate between themselves, (2) it improves the computational complexity as the multiply-and-accumulate (MAC) operation can be replaced by the accumulate operation (AC), and (3) it also reduces the memory overhead as neuron's outputs can be represented only by one bit. In addition, the spiking neuron is a dynamical dynamical, i.e., it has state variable(s) that allows the neuron to record the temporal history of the input. This makes the spiking neuron capable of capturing spatio-temporal patterns at the neuron level, allowing feedforward SNNs to achieve outperforming accuracy in spatio-temporal applications without recurrent connections at the network level [97]. The SNN potentials show how the neuromorphic system (from the neural network level) can be a potential approach for implementing AI tasks with outperforming accuracy, while still requiring a smaller energy and power budget.

This dissertation focuses on the algorithmic level of the neuromorphic engineering stack as highlighted in Fig. 1.2. We aim to develop the SNNs used in well-defined AI tasks, mainly in image classification, while maintaining an optimum energy, latency and cost. However, there are some intrinsic properties in the SNNs that may degrade their processing efficiency. This is discussed in the next subsection.

Application (e.g., dynamic vision [83, 8, 52, 37], edge intelligence [92, 93, 122], medical systems [1, 80])
Algorithm (e.g., network [95, 50] , neuron [41, 11, 77], synapse [7], learning [12, 58], encoding [5])
System Software (e.g.: Lava [102], LCompiler [63], SentryOS [111], Corelet [4])
Hardware Architecture (e.g., Loihi [21], BrainScaleS [81], SpiNNaker [29, 32], DYNAPs [71], TEXEL [33])
Circuits (e.g., analog [71, 109, 40, 65], digital [67, 21, 32],)
Device (e.g., transistor [71, 109, 40, 65], memristive device [74, 69, 90, 54, 70])
Technology (e.g., CMOS [65], ferroelectric [69], phase change [10])

Figure 1.2: Examples of neuromorphic designs across the different abstraction levels of the design stack. This dissertation focuses on neural network designs that are oriented for efficient memory usage.

1.2 Research Problem

In neuromorphic engineering, the design of the neural network is not isolated from considering requirements from the other levels of the design stack shown in Fig 1.2. This is essential to meet two critical requirements that features the neuromorphic system: implementing an AI task at (1) high accuracy and (2) low power and latency. For example, one can extend the applicability of SNNs by naively scaling the width and depth of the network architecture. However, the design of an SNN at the algorithmic level is constrained to not increase the power or the latency compared to an ANN designed for the same AI task.

Most of the work in SNN inherits well-established concepts from machine learning (ML) to train ANNs. On the one hand, this allows the SNN to be applied in many applications such as computer vision [103, 37, 121, 35], speech and audio recognition [115, 116, 23, 117], and other applications [118, 6, 110, 46]. On the other hand, most of these works do not consider the dynamical differences between SNN and ANN, leading to inefficient SNN architectures in terms of latency and power, and surprisingly less efficient than (or at least as efficient as) an ANN of same network architecture implemented on the same computing platform [20]. Even though SNNs are efficient at the network level, they are dynamic and hence complex at the neuron level. In other words, an SNN operates in multiple time steps, where in every step the whole SNN is executed. In addition, every neuron in the SNN has at least one state variable that is kept in memory and updated at every time step. Therefore, naively designing the neural architecture of the SNN results in huge memory overhead. This can be clearly seen when comparing the software implementation of an ANN to an SNN. For example, in ANN of three fully connected layers, the outputs and activations of the first layer are completely evicted from memory once the outputs of the second layer are computed. The same applies to the outputs and activations of the second layer. For an SNN of the same depth, the state variable of every neuron is maintained in memory for all time steps during the execution of any layer. This means that the runtime memory required for the SNN in this neural architecture is at least 3x more than the ANN.

The increased memory overhead of the SNN imposes challenges to design neuromorphic systems that implement an AI task at high accuracy with low power and latency. The memory overhead affects almost all hardware architectures that host a scaled SNN. In neuromorphic systems that rely on von Neumann architectures (e.g., implementing an SNN on software), the additional state variables cause the number of memory accesses to increase (memory read/write operations), which significantly affect the power

and latency. In addition, the increased runtime memory causes a poor utilization of the hierarchical memory, and hence forces more accesses to larger inefficient memories in the memory hierarchy. Similar problems also appear in digital neuromorphic hardware (such as Loihi [21], TrueNorth [2] and uBrain [104]). The large number of neurons in a scaled SNN requires a large number of specialized cores that internally have volatile memory units to keep synaptic weights and neuron's state variables. Therefore, due to the volatility of these memories, this increases the runtime power. In practice, scaled digital hardware platforms, such as intel Hala Point [17] that relies on Intel Loihi chips [21], has a high cost, which is difficult to deploy on resource-constrained computing systems. This cost can be reduced if the computation and their corresponding memory are queued in an external memory system (such as DRAM or FLASH memory). However, this results in power and latency problems similar to those in von Neumann architectures. In analog neuromorphic systems, the memory overhead of SNNs makes their deployment challenging. Analog neuromorphic hardware (such as DYNAP [71] and TEXEL [33]) has a large neuron area compared to digital hardware. This makes designing an analog neuromorphic hardware that is in the same scale as deep SNNs costly. Therefore, implementing a large scale SNN on the state-of-the-art analog neuromorphic hardware requires queuing computations in an external memory system, resulting in similar power and latency problems as in the digital hardware architectures. Therefore, there is a contradicting relationship between the SNNs (neuromorphic system at the algorithmic level) and the host hardware architectures (neuromorphic system at the hardware level). This presents a problem that opposes the main motivation of neuromorphic systems that aims at emulating the biological systems in terms of, both, AI tasks accuracy and processing efficiency.

To address the outlined problem, we need to propose spiking neural networks that align with two main features: high accuracy and efficient processing. In addition, we need to propose solutions that align with the current state of the neuromorphic hardware to support the progress in the field of neuromorphic engineering. On the other hand, neuromorphic hardware is not yet mature for development of scaled SNN and complex applications. Therefore, we need to architect SNNs that can be deployed on a wide range of neuromorphic (and even conventional) hardware architectures with reduced memory overhead. This brings up the following question:

Q1. What is the key design principle that can be deployed on a wide range of hardware architectures and still be memory efficient?

The main design principle is presented in Chapter 3 and the rest of the dissertation builds on this design principle.

The dynamicity of the SNNs also makes them rich in the way the data flows through them and hence makes the memory patterns within the network diverse. Therefore, alleviating the memory overhead in the SNN can be done in different ways. For example, we can reduce the memory requests or improve the data dependencies such that it improves the utilization of a hierarchical memory system.

Q2. What are the possible approaches to reduce the memory overhead of SNNs?

Q3. What is the quantified gain in terms of processing efficiency when reducing the memory overhead of an SNN?

Q2 is addressed by two main approaches: The first approach is adopted in 3 and 4, and the second approach is adopted in Chapters 5 and 6. Q3 is answered by presenting the efficiency gains of each proposed solutions in Chapters 3 to 6.

Another unique aspect of spiking neural networks is that they are dynamic and recurrent at the neuron level. This makes them more flexible in the way they can process the input in the temporal dimension (we have spatial and temporal dimensions to extract features from the input). The biological neuron is also dynamic but still efficient at the same time. This raises the following question:

Q4. What is the relationship between dynamicity and memory efficiency?

Q5. How the dynamicity of the SNN can be utilized to reduce the memory overhead?

These questions were addressed in chapters 5 and 6. Chapter 5 focuses on the utilization of dynamicity of the spiking neuron to do complex vision tasks efficiently, while Chapter 6 introduces utilizes the dynamicity to reduce the memory overhead.

1.3 Research Objective and Methodology

The goal dissertation is to reduce the memory overhead of SNNs, while still maintaining the dynamic properties of the spiking neuron and interfering minimally on the design of the neuromorphic hardware.

The approach adopted focuses mainly on the algorithmic level, that is, the design of neural network architecture. The design of the neural architecture is restricted to the two main metrics of neuromorphic engineering mentioned previously: maintaining the accuracy of the AI task while optimizing their efficiency when running on a hardware platform. As the neuromorphic hardware systems are not mature and available for developing SNN at large scale, we aim to propose neural network architectures that restrict the neuromorphic hardware minimally and, therefore, allowing parallel and independent progress of research at the hardware. In other words, the aim is to propose applicable SNN architectures that are constrained to run at power and latency, while being agnostic to the underlying neuromorphic hardware. In addition, we also aim to be agnostic on the biological plausibility of the network architecture, while using computationally simplified biological neuron models. Even though the dissertation's approach is agnostic with respect to the biological plausibility of the network architecture, the proposed approaches have potential to improve the understanding of neural architectures of biological systems. Although understanding the overlap between neuroscience and the proposed neural architectures is important, this dissertation does not relate the proposed work to neuroscientific research.

The compatibility of the proposed solutions with neuromorphic engineering is measured in terms of applicability and efficiency. Applicability is quantified by the accuracy of the AI task. The applicability of the proposed solutions should not differ from that of the state-of-the-art solutions. In terms of efficiency, we focus on reducing the memory overhead of the SNNs, as the memory overhead is considered as the main efficiency bottleneck in almost every hardware system. This memory overhead includes memory requests (e.g., read and write operations in von Neuman architecture), the number of variables required during runtime (e.g., reflecting the utilization of hierarchical memory system), and the storage requirement (e.g., reflecting the cost of the host hardware; it is the size of memory needed to store the SNN model).

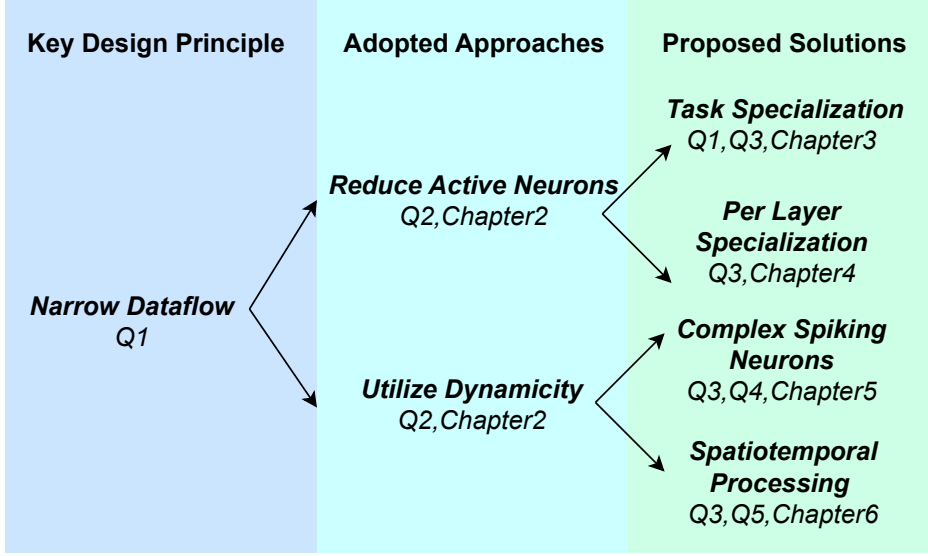


Figure 1.3: Overview of the Contributions

1.4 Dissertation Contribution and Outline

The contribution of this dissertation is to aid in improving the processing efficiency of the neuromorphic system by reducing the memory overhead in the SNNs. This section explains the contribution of each chapter; a summary of the contributions is shown in Fig. 1.3. This section also outlines the rest of the dissertation.

Chapter 2 gives a background on the spiking neural networks. It presents the internal dynamics of commonly used spiking neuron model, the different ways to encode the input and interpret the output, and the training algorithms. The chapter then dives into detail into the memory overhead problem and discusses its impact on the processing efficiency of SNNs. It also presents potential solutions in the literature that could alleviate the memory overhead problem, and accordingly answers Q2.

Chapter 3 discusses further the problem of memory overhead and presents a key design principle to reduce the memory overhead by *reducing the width (narrowing the dataflow)* of the SNNs (answer to Q1). This chapter also introduces the first approach of reducing the memory overhead by *reducing the number of active neurons* in the SNNs (answer to Q2). Then, the chapter introduces the concept of *task specialization* and discusses its potential to reduce the memory overhead of SNNs. Task specialization in this chapter is applied to the classification task, such that a given task is divided into a set of specialized sub-tasks that are used to design memory-efficient SNNs. The proposed solution performs the classification task as a collaborative effort of multiple

specialized and narrow SNNs rather than being performed by a single and fairly large SNN. This contributes to the field of neuromorphic engineering by allowing the use of specialized and ensembled SNNs to design narrow SNNs, which results in reduced number of memory operations and increased utilization of memory hierarchy. In addition, this chapter proposes a workflow that maintains the architecture of a baseline SNN (i.e., maintaining the number of layers as well as the number of neurons per layer), while systematically reducing the dependencies at every layer of the SNN. This maintains the properties of the baseline SNN architecture, while significantly reducing the memory overhead. Task specialization improves the energy and latency of the SNN's inference by at least 10x compared to a non-specialized SNN.

Chapter 4 further adopts the approach of reducing the dynamicity of SNNs (answer to Q2). This chapter discusses the main drawbacks of the task specialization approach and, to address them, proposes the solution of *per layer specialization*. In this solution, every layer is divided into independent specialized experts that are activated by a gating function. This leads to a narrow dataflow, which reduces memory overhead and computations and hence improves the processing efficiency of SNNs. This chapter contributes to the field of SNNs by proposing a systematic and trainable specialization of tasks, which reduces the training burden and eliminates the need for human experts to design the specialized classification tasks. In addition, the accuracy of the tested AI tasks is increased compared to the task specialization solution. The proposed solution in this chapter improves the inference processing efficiency on real general-purpose hardware by up to 3x while maintaining state-of-the-art accuracy.

Chapter 5, adopts the approach of utilizing the dynamicity of the SNNs to reduce their memory overhead (answer to Q2). The dynamical properties of the spiking neuron are studied in detail (answer to Q4). This chapter discusses how the inherent properties of the spiking neuron can be utilized to process an image in a memory-efficient manner. The main contribution of this chapter is to use biologically plausible neuron models, which show interesting capabilities to capture complex temporal patterns. This chapter contributes to the field by presenting an analytical study of the ability of the spiking neuron to extract temporal features. In addition, this chapter proposes the use of the Adaptive Exponential Leaky Integrate-and-Fire (AdExLIF) spiking neuron [11] for efficient extraction of temporal features. This chapter also presents the potential of using AdExLIF to process images temporally. The results in this chapter show that a single AdExLIF neuron is capable of processing a whole image to extract features such as horizontal and vertical edges.

In chapter 6, presents the use of more complex biologically-inspired spiking neurons,

such as the balanced resonate-and-fire (BRF) to reduce the memory overhead of the SNNs in image applications. This chapter exploits the complex temporal dynamics of the spiking neuron to sequentially process an image as a sequence of 2D tokens/patches (answers to Q2 and Q5). This reduces the size of intermediate results in the SNN at the expense of increasing the number of steps. Therefore, this results in flexibility in the SNN, which allows optimizing the processing efficiency based on the available memory resources. The results of this chapter show that the proposed solution reduces the runtime memory requirement for a convolutional SNN by at least 4x and improves the throughput by at least 2x under hardware of constrained memory resources.

Chapter 7 concludes the dissertation by summarizing the dissertation, presenting the main challenges faced during the research process, and finally presenting future research directions and open challenges.

Chapter 2

Background

This chapter provides a background material to the dissertation. First, it starts with background SNNs and the design choices made throughout the dissertation. This includes the spiking neuron model, the input encoding and the output interpretation, the SNN training and finally the computational overhead. Then, the chapter moves to presenting the memory overhead of the SNNs in detail.

2.1 Spiking Neuron Networks

Spiking Neuron Model.

The SNNs are conceptually similar to the conventional ANNs except that they use spiking neurons. As shown in Fig. 2.1, the spiking neuron receives spikes as input ($S^{L-1}[t]$) and produces spikes as output. The temporal dynamics of the spiking neuron is controlled by the spiking neuron model. The leaky integrate-and-fire neuron (LIF) model is a widely used model due to its computational complexity. In the LIF model, the input spikes temporally accumulate in an internal state variable called the membrane potential ($v^L[t]$). If the membrane potential exceeds a threshold value (v_{th}^L), the spiking neuron produces an output spike ($S^L[t]$). The state of membrane potential decays with time using a decaying function; in Fig. 2.1, the decaying function is a linear function with a decaying factor (α).

One can think of the spiking neuron as an accumulator of dot product operations (correlations) between a stored feature vector (the set of weights connected to the spiking neuron) and a set of boolean feature vectors applied sequentially (one vector is applied to the spiking neuron in every time step). In addition, the decaying factor plays a role in forming short- and long- term memories within the SNN [97]. These aspects

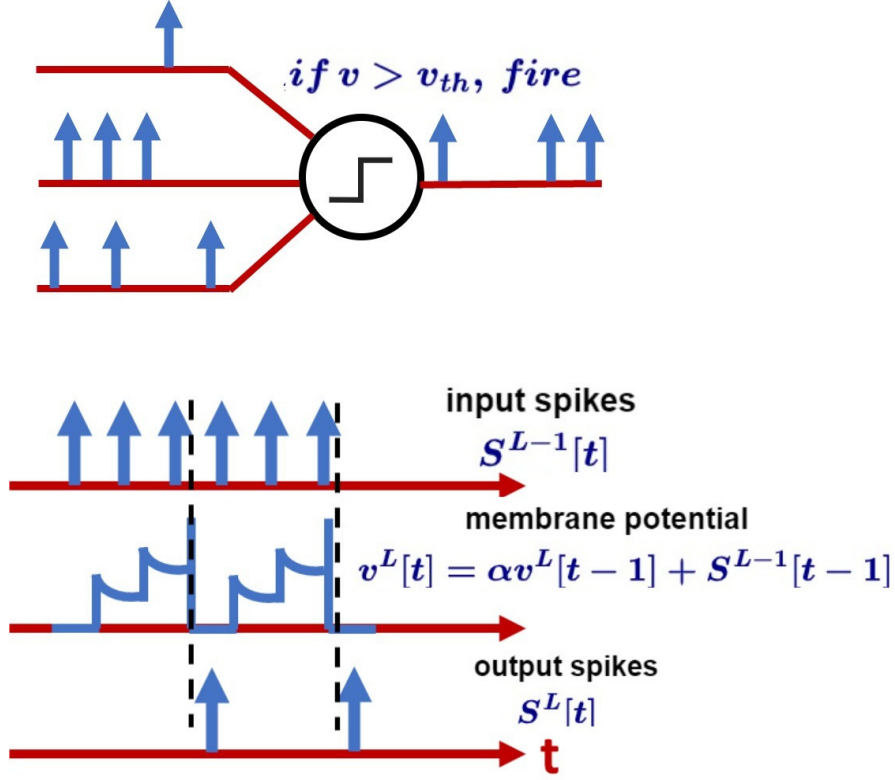


Figure 2.1: The spiking neuron receives spikes as input and produces spikes as output. The input spikes are accumulated in the membrane potential ($v^L(t)$), and an output spike is generated if $v^L(t) > v_{th}$.

allow the SNNs to learn spatio-temporal patterns at the neuron level [97]. In chapter 5, we present more details of the temporal dynamics of the LIF and more complex models.

Input Encoding and Output Representation.

The input and output spikes are used to encode data and extracted information in different ways. This introduces a new dimension in the design of neural networks. In image classification application, each pixel in the input image is encoded as a sequence of spikes. Common examples of input encoding are rate encoding, delay encoding, and direct encoding [5]. In rate encoding, each pixel value is encoded into multiple one-bit values (spikes). The frequency of these spike (spiking rate) encodes the intensity of every pixel. Delay encoding encodes each pixel value into two spikes of specific temporal distance. For example, if the pixel value is higher, the temporal distance between the two encoding spikes is larger. This type of encoding is energy efficient as it uses only

two spikes per pixel. Finally, direct encoding applies the whole input image as it is, while the first layer automatically encodes the pixels to spikes. In this dissertation, we adopt the use of direct encoding as it produces higher accuracy in image classification applications [53].

The output of an SNN can also be interpreted differently. In image classification tasks, one output spiking neuron is dedicated per class. The output neuron that fires more indicates the classification output. Other approaches are also used to interpret the SNN output. For example, in an SNN with delay input encoding, the output is indicated by the neuron that fires last. Another common example is to use the membrane potential to train the SNN. The membrane potential is often used in regression tasks due to its continuous nature compared to the discrete spiking output. As this dissertation focuses on image classification tasks with direct encoding, we adopt the most firing neuron approach to interpret the output.

SNN Training.

Backpropagation through time (BPTT) [114, 9] is commonly used for supervised training of SNNs. However, training of SNNs using BPTT is more challenging than ANNs. The Heaviside activation function used by the spiking neurons vanishes the gradients during backpropagation. Surrogate gradient descent (SGD) replaces the derivative of the Heaviside function with a smoother function during the backward step [76, 26]. There are also other techniques that can be used to train the SNNs such as the biologically-inspired learning, i.e., spike-timing-dependent plasticity (STDP) [30] or ANN-to-SNN conversion techniques [39, 120]. These techniques either work with shallow SNNs (such as the STDP learning) or produce SNNs with inefficient spiking activity (such as ANN-to-SNN conversion) [26]. Therefore, we adopt in this work the use of BPTT to train the SNNs.

Computational Overhead of the SNNs.

Computationally, the SNN has two main defining features: (1) the spiking communication between neurons and (2) the dynamicity of the neuron. These two features result in conflicting effects in terms of processing efficiency.

At the network level, the spiking neurons communicate only using spikes. This enables efficient implementation of the SNNs on hardware. First, the multiply and accumulate (MAC) operation, which is used to modify the output of a network layer, can be replaced by the accumulate operation (e.g., as in [47]), which improves the

processing efficiency regardless of the architecture of the host hardware. Second, spiking communication opens up a new design space for asynchronous hardware (at the network level) [22, 104], which facilitates the design of efficient custom hardware.

At the neuron level, the SNNs are dynamical, where the spiking neuron has one or more internal state variables. This makes spiking neuron a system by itself, in which it may have varying complexity based on its biological plausibility. For example, the Hodgkin-Huxley model exhibits richer temporal dynamics compared to the integrate-and-fire (IF) model, whereas it is computationally more complex. This trade-off between the richness of temporal dynamics and the computational complexity of the neuron model offers a new abstraction in the design of the neural networks. In addition, the temporal dynamics enables the spiking neuron to extract features at the neuron level. For example, some work utilizes the complexity of the spiking neuron to extract spectral features [77, 66], which is not feasible with conventional static neuron such as ReLU and Sigmoid.

In summary, SNNs improve the processing efficiency of neural networks at the network level, while they degrade the processing efficiency at the neuron level. This shows that the SNN requires a different design approach to balance this conflict in efficiency. In chapter 5 and 6, we show some design tradeoffs when utilizing the temporal dynamics differently in image applications. The next section focuses the processing inefficiencies that the spiking neuron’s dynamicity introduces.

2.2 Memory Overhead of SNNs

The dynamicity of the spiking neuron adds computational and memory overheads in comparison to a static neuron. The computational complexity can be mitigated by utilizing the temporal dynamics of the neuron to extract features efficiently. For example, a spiking neuron can be seen as a non-linear infinite impulse response (IIR) system [27, 94, 77], which is capable of extracting features with less operations and parameters compared to finite impulse response (FIR) systems. On the other hand, the dynamicity of the spiking neuron introduces a significant memory overhead, which may diminish the computational benefits of the spiking representation at the network level.

To gain insight into this memory overhead, we compare the software implementation of a feedforward ANN and SNN. In a feedforward ANN, all intermediate outputs produced by any layer can be completely removed from the memory after producing the output of the next layer. In comparison, SNNs have dynamic membrane potential variables that evolve over time. This property requires maintaining a temporal history

Algorithm 1: Maintain membrane potentials for every neuron in SNN for all time steps

```

1 for every step  $n$  do
2   for every layer  $L$  do
3     // Copy the inputs of  $L$  (into the tensor  $S_{in}$ )
4     // from the outs of  $L-1$  (from the tensor  $S_{out}$ )
5      $S_{in} = S_{out}$ 
6     //  $S_{out}$  is the outs tensor of  $L$ ;
7     //  $mem$  is the mem. pot. tensor of all layers
8      $S_{out} = \text{SpikingFunction}(S_{in}, mem[L])$ 
9     // Update  $mem[L]$ 
10     $mem[L] = \text{MemUpdate}(S_{out})$ 

```

Algorithm 2: Compute and maintain the output of the current layer for all time steps

```

1 for every layer  $L$  do
2   // Copy the inputs of  $L$  (into the tensor  $S_{in}$ )
3   // from the outs of  $L-1$  (from the tensor  $S_{out}$ )
4    $S_{in} = S_{out}$ 
5   for every step  $n$  do
6     //  $S_{out}$  is the outs tensor of  $L$  for all steps
7     //  $mem$  is mem. pot. tensor of  $L$ 
8      $S_{out}[n] = \text{SpikingFunction}(S_{in}[n], mem)$ 
9     // Update  $mem$ 
10     $mem = \text{MemUpdate}(S_{out}[n])$ 

```

of the SNN execution implicitly (by maintaining the membrane potential variables in memory) or explicitly (by maintaining the outputs of the currently executed layer for all time steps). Algorithm 1 and Algorithm 2 show two different execution paradigms of the SNN inference. In Algorithm 1, the membrane potential variable of every neuron in the SNN is maintained in the memory even after executing every layer, while the output tensor of every layer is removed from memory once the next output tensor is computed. In Algorithm 2, for every layer, the output tensor is computed for all time steps and maintained in the memory, while the tensor of membrane potentials is removed from memory once the output tensor is computed for all time steps. During the inference, both implementations impose an extra memory overhead compared to the conventional feedforward ANNs.

To quantify this memory overhead, Fig. 2.2 shows a comparison between SNNs and ANNs (with convolutional layers) in terms of the maximum runtime memory (measured in software implementation using PyTorch). The SNN is implemented with Algorithm 1. The runtime memory of the SNN increases significantly with increasing the number of layers when compared to the ANN. In the ANN, the outputs of the processed layers are removed from memory once the next layer is processed. In the SNN, the values of the state variables of a layer must be maintained in memory, even if the outputs of

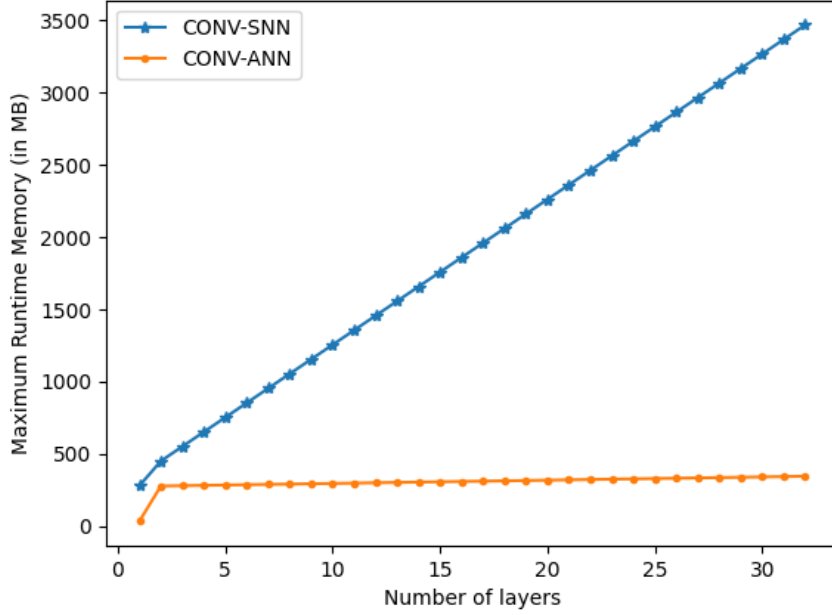


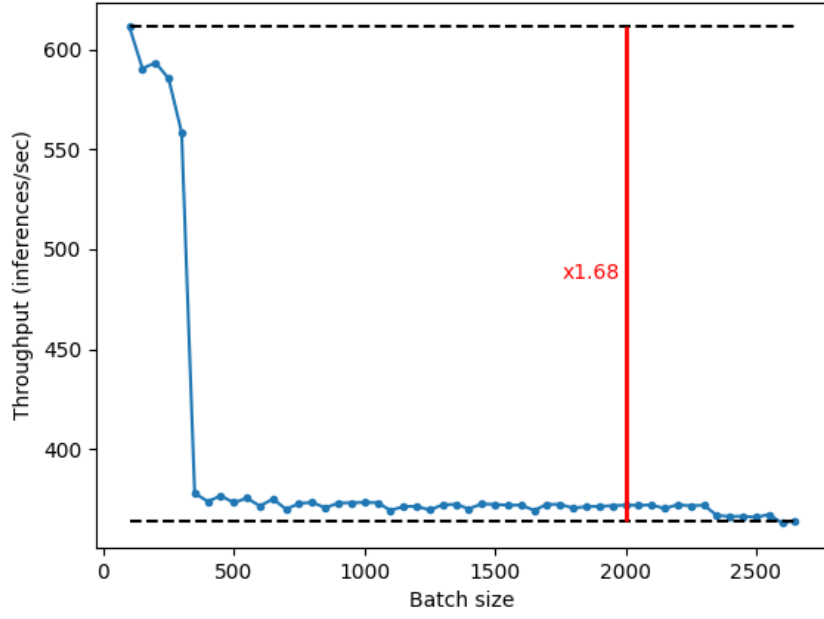
Figure 2.2: Maximum runtime memory for different numbers of layers

this layer are used in the processing of the next layer. This constraint in maintaining the state variables in SNN is required to use them in subsequent time steps (i.e., the spiking neuron is dynamic).

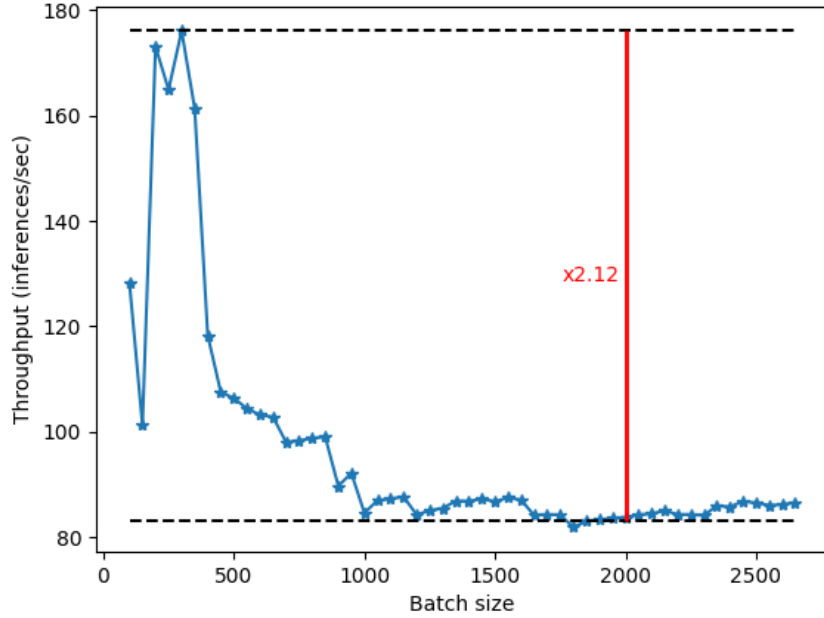
To evaluate the impact of this increased memory in the SNN, we show in Fig. 2.3 the throughput (inferences per second) when the ANN and the SNN are implemented on a CPU with a hierarchical memory. In both networks, when increasing the batch size, the required memory pushes the hardware towards its communication bottleneck. As can be seen in the figure, the degradation of the throughput is higher in the SNN when compared to the ANNs. This shows how the additional memory overhead introduced in the SNN degrades the processing efficiency more than the ANN.

In summary, the additional memory overhead of the SNN results in two main problems:

1. **Degraded processing efficiency:** With a limited working memory (e.g., the main memory in a conventional computing system or the on-chip memory of a neuromorphic system), the neuron’s state variable of the SNN need to be moved to a denser storage device (e.g., FLASH memory). This results in more frequent accesses to the storage device during runtime, and hence resulting in inefficient read and write operations. This increases the overall energy consumption and



(a) ANN



(b) SNN

Figure 2.3: Comparison between ANN and SNN in the utilization of a hierarchical memory system. As both networks are performed layer by layer, an increased batch size pushes the memory system closer to its communication bottleneck; i.e., intermediate results in every layer does not fit the intermediate memories (caches).

2.3. Exiting Solutions to Alleviate the Memory Overhead Caused by the SNN’s Dynamicity

latency of the SNN inference, which squanders the other computational benefits of the SNN.

2. **Increased hardware cost:** As we need to store all membrane potential variables in some form of memory (either working memory or a storage device), we increase the memory requirements of the computing system. Therefore, the overall cost of the computing system increases. This is especially obvious when we desire to increase the size of the working memory. For example, intel Loihi 2 neuromorphic chip has 128 asynchronous neural cores, each including up to 128 KB of synaptic memory and up to 8192 neurons. This limited on-chip memory requires partitioning of large scale SNNs into smaller computations to fit the working memory of the chip [91]. This requires either increasing the on-chip neural capacity, using multiple neuromorphic chips, or using an external memory and sequentially executing the partitioned computations. All these solutions increase the overall cost, which is undesirable for applications with severe constraints on computational resources.

2.3 Exiting Solutions to Alleviate the Memory Overhead Caused by the SNN’s Dynamicity

Solutions were proposed in the literature to alleviate the memory overhead due to the dynamicity of the SNNs. Table 2.1 groups these solutions into five main categories: (1) quantization (that reduces the bit-size of the membrane potential variable, e.g., [85, 84, 124, 113]), (2) pruning (neurons) (that reduces the number of membrane potential variables in the SNN, e.g., [100, 13]), (3) reducing activity (that reduces the number of spikes in the SNN e.g., [73, 105, 52, 60, 53]), (4) reducing the number of steps (e.g., [61, 98]), and (5) membrane potential sharing (that reduces the number of membrane potential variables in the network by sharing, e.g., [48]). Each group was analyzed and evaluated (for the SNN’s inference) by four qualitative metrics; we only compare these solutions qualitatively as the literature lacks a widely accepted benchmark for memory overhead solutions. The first metric is the *reduction in the number of membrane potential variable*. Reducing the number of membrane potential variables reduces the dynamicity potential in the SNN and hence reduces the memory overhead, especially in terms of the runtime memory requirements and also the utilization of a hierarchical hardware memory. The second metric is the reduction in the number of accesses to the membrane potential variables. Accessing the membrane potential vari-

Table 2.1: Comparison of the solutions in the literature that aims at alleviating the memory overhead due to the dynamicity of the SNNs. These solutions are classified under 5 classes: Quantization, pruning, reducing activity, reducing the number of steps, and membrane potential sharing.

Solution Category	Type of improvement	Reduce the number of membrane potentials (eliminate dynamicity)	Reduce the number of accesses to membrane potential	Reduce the runtime memory requirement	Utilize the dynamicity to reduce memory overhead
Quantization [85, 84, 124, 113]	Reduce the bit-size of every membrane potential	No	No	Yes	No
Pruning (Neurons) - e.g., [100, 13]	Eliminate neurons and their corresponding weights	Yes	Yes	Yes	No
Reducing Activity - e.g., [73, 105, 52, 60, 53]	Reduce the number of updates in membrane potential	No	Yes	No	No
Reducing the Number of Steps - e.g., [61, 98]	Reduce the number of updates in membrane potentials	Yes	Yes	No	No
Membrane Potential Sharing - e.g., [48]	Reduce the number of updates in membrane potentials	Yes	Yes	Yes	No

ables less frequently reduces the number of memory operations (read/write) which may potentially lead to better utilization of a hierarchical hardware memory system. The third metric is the direct reduction in the runtime memory requirements. Reducing the runtime memory requirements leads to lower the hardware cost and may also lead to a better utilization of a hierarchical hardware memory. The last metric is the utilization of dynamicity to reduce memory overhead. In this metric, we look for smart solutions that view the dynamicity itself as a potential property that reduces the memory overhead rather than causing it.

Each category listed in Table 2.1 contributes to reducing memory overhead; however, they target different aspects of the memory overhead. The solutions based on quantization and reduction of the spiking activity do not reduce the number of membrane potentials. These solutions reduce the memory overhead by either simplifying the bit-level representation (in quantization) or reducing the number of memory operations. Maintaining the number of membrane potentials allow for maintaining the dynamical complexity of the SNN. In terms of the number of accesses to the membrane potentials, only quantization-based solution do not reduce the number of accesses to the membrane

2.3. *Exiting Solutions to Alleviate the Memory Overhead Caused by the SNN's Dynamicity*

potential. This shows that most solutions in the literature (including the quantization-based solutions) improve the utilization of hardware memory bandwidth (utilization of hierarchical memory). Reducing the runtime memory is another important metric that reflects the hardware cost and the utilization of a hierarchical memory. Some categories improve reduce the runtime memory requirements such as the quatization, pruning and membrane potential sharing based solutions. Finally, up to our knowledge in the literature, there is no found solution that considers the dynamicity of the SNN to reduce the memory overhead.

This classification allows to see a gap in the literature in two important aspects: First, the available solutions in the literature seem to tradeoff an improvement in one aspect of the memory overhead for another aspect or by eliminating dynamicity in the SNN. For example, no single solution in the literature improves the memory overhead in terms of reducing the number of memory accesses and the runtime memory requirements without compromising the dynamicity of the SNN. This means that if an approach improves the memory overhead of the SNN in all aspects, it reduces the SNN's dynamicity which implies pushing the SNN's functional complexity closer to the static ANNs. **This brings us to an answer to Q2:**

Recall: Q2. What are the possible approaches to reduce the memory overhead of SNNs?

Our first approach to reduce the memory overhead of the SNNs is to **reduce the number of active neurons in the SNN** without sacrificing the dynamicity. This approach is adopted in chapters 3 and 4.

The second gap that can be observed from Table 2.1 is that the inability of the approaches in the literature to utilize the inherent dynamicity of the spiking neuron to reduce the memory overhead. In other words, most solutions in the literature borrow concepts from the field of ANN and apply to the SNNs. This allows us to reduce the memory overhead in the SNN but not providing a fundamental solution that relies on the intrinsic properties of the SNNs. **This brings up the second approach in this dissertation (answer to Q2):**

Our second approach to **utilize the dynamicity of the spiking neuron to reduce the memory overhead** rather than viewing it only as a cause of the memory overhead. This approach is adopted in chapters 5 and 6.

In the next chapter, we will start with the first proposed solution in the dissertation and also set the key design principle employed throughout the whole dissertation.

2.3. Exiting Solutions to Alleviate the Memory Overhead Caused by the SNN's Dynamicity

Chapter 3

Task Specialization for Efficient Processing of Spiking Neural Networks

This chapter we present and justify the key design principle used throughout the dissertation (Q1). It also presents our first solution and quantifies the gain in processing efficiency while reducing the memory overhead of the SNNs (Q3). The proposed solution performs the classification task as collaborative tasks of specialized SNNs. This specialization reduces the number of memory operations and improves the utilization of memory hierarchy. In addition, task specialization reduces the number of active neurons (and hence the runtime memory requirements) by executing narrow SNNs sequentially. The results show that task specialization improves the energy and latency of the SNN's inference by more than 10x. Task specialization shows that designing narrow (and deep) SNNs is *computationally more efficient* than designing wide (and shallow) SNNs. The content of this chapter was published in the following paper: Muath Abu Lebdeh, et al. "Efficient processing of spiking neural networks via task specialization", IEEE Transactions on Emerging Topics in Computational Intelligence, 8(5):3603–3613, 2024.

3.1 Introduction

The previous chapter discussed the importance of reducing the memory overhead of the SNNs. In this chapter, our aim is to reduce the memory overhead of the SNNs by reducing the number of memory operations and increasing the utilization of memory hierarchy. The proposed approach to reduce this memory overhead relies on the specialization of the classification task into simpler subtasks. There are several related works

in the literature that either (1) share a similar purpose of reducing SNNs’ computational overhead in general and the memory overhead in specific while using different and orthogonal approaches, (2) share a similar approach of using an ensemble of weak SNN classifiers but not directly to improve the processing efficiency of the SNNs, or (3) share a similar approach of using specialized networks for reducing the ANNs computations but not for reducing the memory overhead of the SNNs. This section discusses these three types of related work.

Reducing the memory overhead of SNNs. Our work has a very similar goal to the work in [48]. This work aims at reducing the number of membrane potentials at any time step, which automatically decreases the memory requirements of the SNNs, the number of memory operations as well as the utilization of the memory hierarchy. Although this work has the same goal of our work, we take a different direction to reduce the memory overhead. We reduce the memory overhead via specialized ensembles of SNNs, which does not overlap with the approach in [48]. In other words, these approaches are orthogonal and can be integrated; e.g., the approach in [48] can be applied to the specialized SNNs and further reduce their memory overhead.

SNN ensembles. The work in the literature that used ensembles of SNNs have different goals than decreasing the memory overhead of SNNs. In [75], SNN ensembles were used to facilitate the training of SNNs, and decrease the overall number of parameters. Even though this work improves the processing efficiency of SNNs, it still does not directly address the high memory overhead in SNNs. In addition, the datasets tested in this approach are relatively simple. In [87], two unimodal SNN ensembles were used to enable multimodalities in SNNs with STDP training. The focus of this work is to enable a new application in SNNs rather than improving the processing efficiency of SNNs. In [24], an ensemble of weak SNN classifiers were used to reduce the overhead of extracting SNN architecture using evolutionary algorithms. This work does not aim to improve the processing efficiency of SNNs; in fact, they use an ensemble of 50 SNNs to implement a toy application (UCI handwritten digits). In [99], HybridSNN was proposed as an adaptable architecture that is made of a homogeneous ensemble of SNNs, ranging from single-layer SNNs to convolutional SNNs. The goal of this approach is to stabilize and facilitate the training of SNN (in a modular and biologically plausible way). Although this approach also can improve the processing efficiency of SNNs, it does not directly target reducing the memory overhead of SNNs. The HybridSNN can learn architectural patterns with wide layers (layers with large number of input or output features) and hence increased memory overhead.

Sparsely-gated mixture-of-experts (SG-MoE). Our work has a similar (but sim-

pler) approach to the work of SG-MoEs [96, 89, 14] while targeting a different goal. These papers aim at skipping computations and hence increase the processing efficiency of ANNs. Although this also can be valid for our approach, we aim at utilizing the notion of expertise to systematically decrease the width of the specialized SNN models, and hence reduce the memory operations and increase the utilization of memory hierarchy. Note that with some modifications to the training in our approach, we can also utilize the same approach for a novel computation skipping in SNNs. In addition, the work that deploys the SG-MoE in image classification applications relies on vision transformers [89], such that the input image is applied sequentially as patches. Each patch can be processed by a small set of models from the whole set of MoEs (e.g., some patches visually present the background, other patches present specific geometrical shapes, etc). In our work, we implement the specialty in the SNNs without patching the input image, such that we consider the whole input image as a patch that can be processed by a specialized SNN.

Other general approaches to improve the processing efficiency of SNNs. A group of work re-used the network compression techniques via pruning and quantization for SNNs [112, 51, 49]. These techniques reduce the model size by reducing the neurons count (e.g., eliminating neurons with low firing rates), reducing the synaptic connectivity, or reducing the bit size of the network’s parameters. These compression techniques still do not push the energy and latency of an SNN to a fundamentally low level; for example, unstructured pruning is a common technique used to remove redundancy, while it does not fully benefit from sparse tensor accelerators [79] which require structured input data (matrices and vectors). Therefore, compression techniques are mainly useful for improving the storage and memory requirements of the SNNs, which indirectly improve the energy and latency.

Another group of solutions penalizes the SNN during training to reduce the spikes count [123, 73]. These solutions allow the SNNs to make better utilization of sparse and asynchronous computing hardware. However, as our results show (in section 3.3), reducing the spikes count alone does not necessarily reflect the efficiency of processing an SNN. The energy consumption and latency depend on both the spikes count per layer and the width of the next layer.

Other work targets reducing the number of time steps [19, 50, 62]. These techniques are especially useful for SNNs converted from ANNs. Reducing the number of time steps is equivalent to reducing the sequential sub-inferences required for complete SNN inference. As fewer sequential sub-inferences are required, this improves the overall energy and latency. On the other hand, although reducing the time steps improves

the efficiency of a specific SNN, time steps alone is not a metric for latency as well as energy. For example, an SNN with low spiking activity and a high number of steps can be faster and more energy-efficient than an SNN with a low number of time steps but high spiking activity.

Another promising group of techniques utilizes special encodings to reduce the energy and latency of SNN [52, 53, 59, 60]. These encodings rely on using a sparse representation of the input data to reduce the overall spike count in the SNN. For example, in delay encoding, each input feature is represented by two spikes, where the amplitude of the feature is represented by the number of time steps between these two spikes. On the other hand, this low number of spikes in the input results in vanishing outputs in deep layers of SNNs, which makes training deep SNNs difficult [52]. Moreover, representing information using a low number of spikes makes the inference more prone to error [52, 53].

In this chapter, we propose utilizing the idea of task specialization to design narrow SNNs, and hence reduce the memory overhead of SNNs. This reduction in memory overhead includes reducing the number of memory operations and increasing the utilization of memory hierarchy. The proposed approach performs the classification task as a collaborative effort of multiple specialized and narrow SNNs rather than being performed by a single and fairly large SNN. We show that training each specialized SNN on a relatively unique subset of the dataset leads to narrower layers and, in turn, decreases the spatial dependencies between consecutive layers in SNNs. This reduction in spatial dependencies reduces the number of memory and computing operations. In addition, the decrease in spatial dependencies allows the specialized SNNs to fit better into more local memories, which reduces the movement of data in different levels of the memory hierarchy. Therefore, our proposed approach reduces the overall inference energy and latency.

Briefly, the contribution of this chapter is summarized below:

- Using the concept of specialized and ensembled SNNs to design narrow SNNs, which results in reduced number of memory operations and increased utilization of memory hierarchy.
- Proposing a workflow (based on specialization) that maintains the architecture of a baseline SNN (i.e., maintaining the number of layers as well as the number of features), while systematically reducing the dependencies at every layer of the SNN. This maintains the properties of the baseline SNN architecture, while significantly reducing the memory overhead.

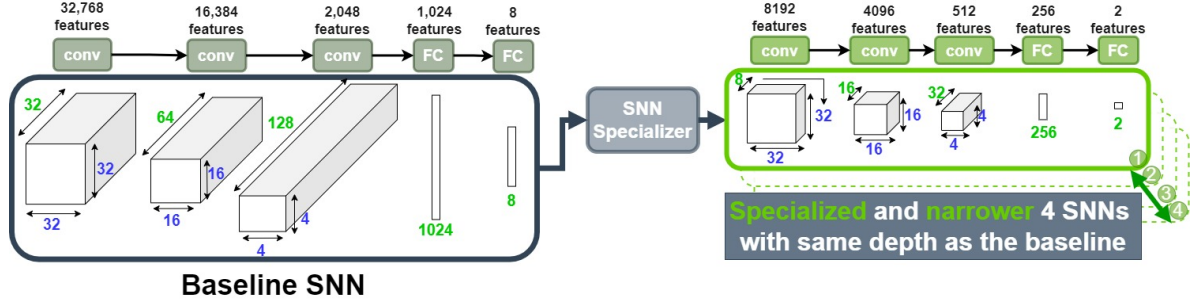


Figure 3.1: A baseline SNN is replaced by multiple independent SNNs of narrow widths. The depth of each narrow SNNs is the same as the baseline SNN. The sum of the features in all narrow SNN at a specific layer equals the number of features at that layer in the baseline SNN. This property allows both structures to have similar memorization and generalization capacities.

3.2 Proposed Solution: Specialized Spiking Neural Networks

This section presents the main parts of the proposed approach including: reducing SNN width via specialization, the design of specialized tasks, training and structuring the specialized SNNs, and aggregating their results. Finally, this section presents the potentials of specialization in reducing the memory overhead of SNNs.

3.2.1 Reducing the width of the SNN model

The problem of wide SNNs. Powerful SNNs that target image classification tasks are usually deep and *wide* [52]. Wide layers are capable of memorizing more features, which gives the backpropagation algorithm a more relaxed optimization space to find a solution. On the other hand, a wide layer has larger dependencies that result in more memory operations and increased data shuffling within the memory hierarchy.

Replacing the wide SNN with narrow SNNs. Motivated by this, we propose replacing a baseline SNN with multiple independent SNNs each with a narrower width than a baseline SNN as shown in Fig. 3.1. This allows us to significantly reduce the number of memory operations as well as to better load the computations to more local memories. The proposed approach maintains the total number of features. For example, the number of features (neurons) of the first layer in the baseline SNNs is maintained across the first layers of all narrow SNNs. Moreover, the proposed approach maintains the generalization capacity by maintaining the depth of the baseline SNN; i.e., every narrow SNN has the same depth as the baseline SNN. These two aspects in

3.2. Proposed Solution: Specialized Spiking Neural Networks

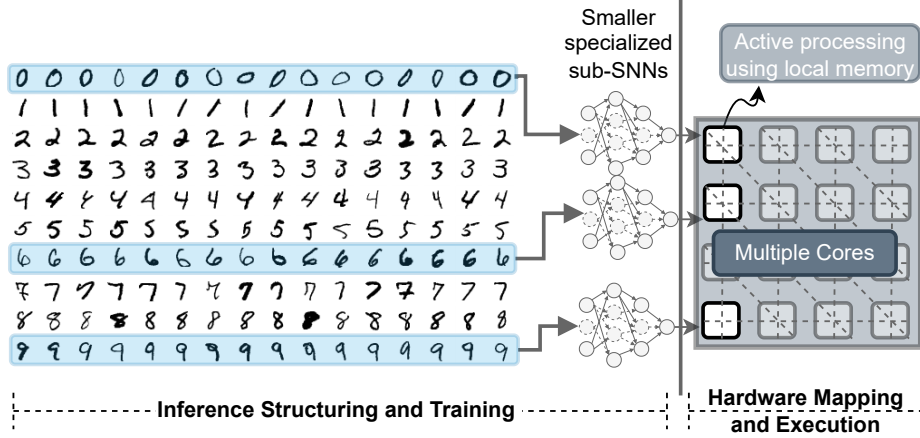


Figure 3.2: The SNN inference is performed as a collaborative effort of multiple specialized SNNs. These SNNs can be mapped to local computing resources, such as the multiple cores presented in this figure. Specialized and small SNNs can be executed sequentially or in parallel.

the proposed approach allow us maintaining the structural properties of the baseline SNN.

Recall: Q1. What is the key design principle that can be deployed on a wide range of hardware architectures and still be memory efficient?

The key design principle used throughout the dissertation is *designing narrow SNNs*. As explained above, the width of the SNN is critical when it comes to the memory overhead. Therefore, designing narrow SNN allows to reduce the memory overhead from different aspects. In addition, as this key design principle is hardware-independent, it allows the proposed solutions to be deployed in a wide range of hardware architectures.

Task specialization. On the other hand, narrowing the width of an SNN reduces the memorization capacity of the SNN, which makes the training more difficult. This challenge is alleviated by simplifying the classification task, which relaxes the memorization requirements. Therefore, we split the image classification task into multiple simpler tasks, each of which is executed by a narrow specialized SNN as shown in Fig. 3.2. We refer to each specialized SNN as a *sub-SNN* which performs a sub-task. The next subsection presents more details about the design of the specialized sub-tasks.

3.2.2 Designing the Specialized Tasks

We specialized every sub-SNN to detect one class in the image classification application, i.e., each sub-SNN is a binary classifier for one class. Image classification applications usually have classes that are mutually-exclusive, such that every class corresponds to one visual object. Based on this, there must be a level of independence between the visual features represented by every class, especially at deeper layers ([101] suggested that the mutual information between the output layer and a hidden layer increases with depth). Conventionally, a single SNN is used to classify all classes. This imposes every layer to learn the specific features of every class. During the inference, this imposes computing all features corresponding to all classes, while the input image belongs to one class. Instead of that, we use a specialized SNN (sub-SNN) for each class in the image classification problem. Therefore, each sub-SNN will learn the most relevant features from the images of the corresponding class. This allows us to remove all unnecessary computations for irrelevant features (the features for other classes) with all corresponding dependencies. Therefore, we reduce the overall memory overhead (less memory operations and better utilization of the memory hierarchy).

As we specialize based on the number of classes, the dataset is divided based on the number of classes. For example, if the dataset contains 10 classes, we re-organize the dataset into 10 specialized sets of data. Each specialized set (the specialized task) is used to train one of the sub-SNNs. Each specialized set is created by resampling the original datasets by over-sampling the target class while under-sampling the rest of classes. For example, if a specialized set is used to train the sub-SNN for detecting class 1, the samples corresponding to class 1 equals to 50% of the total samples, while the other 50% of the samples are sampled equally from other classes. These specialized sets of data is used to train the specialized sub-SNNs. More details on training the sub-SNNs are discussed in next subsection.

3.2.3 Training via Heterogeneous Batching

As described in the previous subsection, each sub-SNN is trained by a specialized set of data. This specialized set of data contains two classes of samples: (1) the target class samples and (2) the samples corresponding to the other classes or the alien class samples. For example, if we have CIFAR10 dataset as the baseline dataset, one sub-SNN is specialized in detecting the airplane object. The corresponding specialized set of data contains: (1) The target class data which contains samples of the airplane object, while (2) the alien class data contains samples from other objects (i.e., automobile,

bird, cat, etc). One way to organize the training batches in this specialized set of data is by sampling 50% of the samples from the target class and the other 50% from the alien class.

On the other hand, this way of sampling the training batches is not effective with small batch size. In fact, the new distribution of samples makes training unstable; every (small) batch contains very few samples from each object in the alien class, which results in a relatively high variance between different samples. This makes the accuracy difficult to converge to acceptable values. Therefore, we need a technique to sort the training data in a way that reduces the variations between different batches.

Larger batches. One way to reduce the variations between epochs is by increasing the batch size so that it contains enough samples from every object in the alien class. However, large batches cause the loss to converge to a local minimum which results in low accuracy.

The sum of weighted gradients. Another way is to weight and accumulate the gradients of more samples in the alien class. In this approach, every batch contains equally distributed samples from every object (e.g., 32 samples for each object). After that the following steps are applied: (1) the gradient is computed for all samples (32x10 samples), (2) the gradients corresponding to each object are averaged, (3) the gradients corresponding to the alien class are averaged, (4) the final values are summed and the sub-SNN parameters are updated. This allows every sub-SNN to be trained equally on all objects from the alien class, even for small batch sizes. On the other hand, computing the sum of weighted gradients is time-consuming; e.g., a conventional batch of 64 samples is scaled to 320 in this approach. We address this problem by proposing the heterogeneous batching.

Heterogeneous batching. As shown in Fig. 3.3, we adopted a hybrid solution that is a composite of the aforementioned approaches. Instead of the sum of weighted gradients, we train using heterogeneous set of batches. Let B_1 be the first set of batches of size N . B_1 is composed of 50% of the samples from the target class and the other 50% from the alien class. This is mathematically formulated as following:

- D_{target} is the set of the target class samples (e.g., the airplane object in CIFAR10).
- D_{alien} is the set of the alien class samples (e.g., the other nine objects in CIFAR10).
- $S_{target} \subseteq D_{target}, |S_{target}| = \frac{N}{2}$ (sampled uniformly at random)
- $S_{alien} \subseteq D_{alien}, |S_{alien}| = \frac{N}{2}$ (sampled uniformly at random)

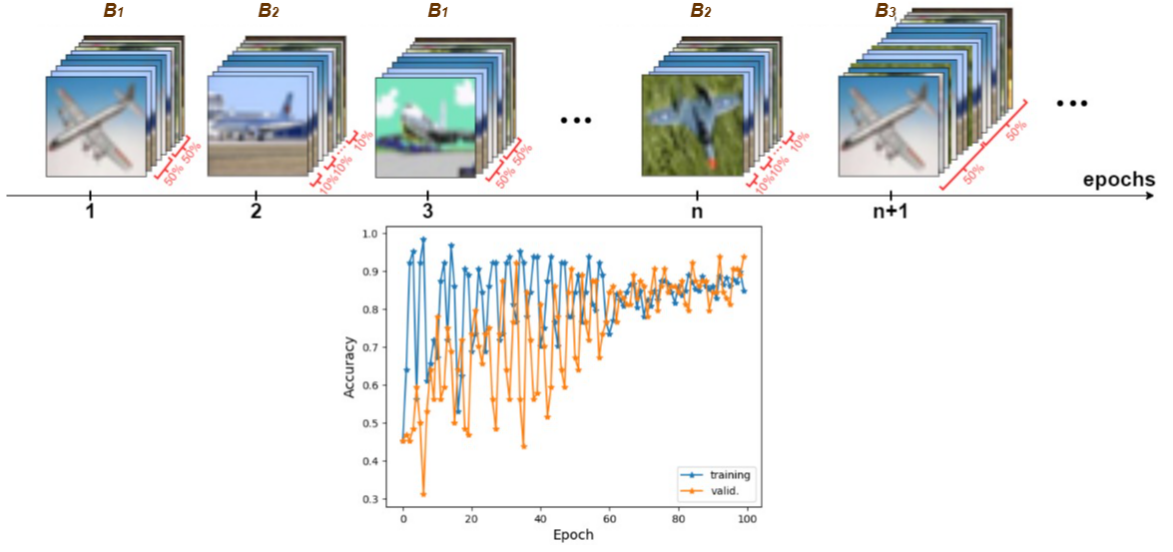


Figure 3.3: Training each sub-SNN with heterogeneous sets of batches. Set 1 contains 50% of the samples from the target object and the other 50% from all alien objects. The samples in set 2 are sampled with equal percentages from all objects. Set 3 is the same as Set 1 except that this set contains larger batches.

- $B_1 = S_{target} \cup S_{alien}$

The second set of batches B_2 is composed of samples equally sampled from all classes. B_2 has more samples corresponding to the alien class. Therefore, the gradients of this batch is scaled by $1/c$ (c is the total number of objects in both classes). This is mathematically formulated as following:

- D_k is the set of object k samples (e.g., class k in CIFAR10 dataset).
- $S_k \subseteq D_k, |S_k| = \frac{N}{c}$ (sampled uniformly at random)
- $B_2 = \bigcup_{i=1}^c S_i$
- After computing the gradients from B_2 batch, they are scaled by $\frac{1}{c}$.

The training is performed by alternating between B_1 and B_2 at the epoch level rather than at the iteration level; i.e., B_1 is applied at odd epochs while B_2 is applied at even epochs. Finally, as shown in Fig.3.3, alternating between different batches produces fluctuations in the learning curve. Therefore, a new set of batches (B_3) is applied. B_3 is the same as B_1 except that it has a larger size (larger N). B_3 starts to be used once the loss reaches a desired value until the end of training. This stabilize the learning curve which allows the loss/accuracy to slowly converge.

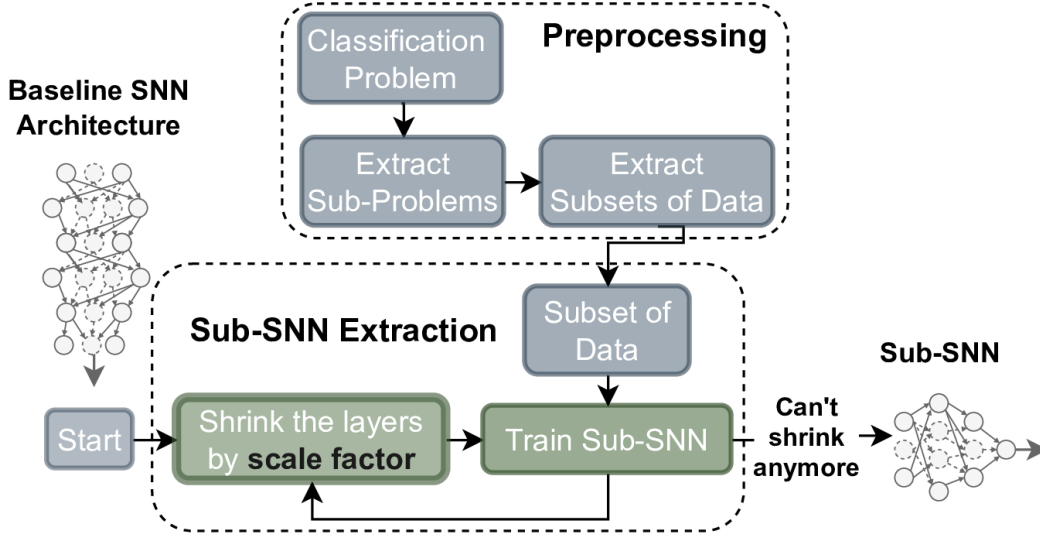


Figure 3.4: A sub-SNN is structured by reducing the width of the baseline SNN, then training it on a subset of the dataset, then comparing the sub-SNN accuracy to the baseline SNN. This process is repeated until the scaled sub-SNN does not meet the accuracy of the baseline SNN.

The proposed heterogeneous batching approach trains the sub-SNNs in a faster time than the weighted sum of gradients while reaching a similar accuracy.

3.2.4 Sub-SNN Structuring

A classification task of C classes is divided into C sub-classification tasks, each of which is executed by a sub-SNN. The baseline SNN architecture is split into C sub-SNNs where each sub-SNN is specialized in detecting only one class. Fig 3.4 shows the workflow of structuring each sub-SNN.

Initially, a given baseline SNN architecture is trained on the whole dataset. The purpose of training this baseline SNN architecture is to reuse the trained weights when creating the specialized sub-SNNs. Each sub-SNN is initially structured by halving the width of every layer in the baseline SNN architecture (i.e., a scale factor of 2). This is done by removing half of the feature maps in every layer. To be able to benefit from the trained weights in the baseline SNN, halving the width is implemented by (permanently) zeroing the weights or the convolution kernels corresponding to the feature maps with the least activity. This ensures that the most relevant features to the specialized task is maintained. If a desired accuracy is reached, the baseline SNN architecture is replaced with the new SNN architecture of halved width. After that, the same process is repeated until we reach an SNN architecture that cannot have narrower widths but maintains

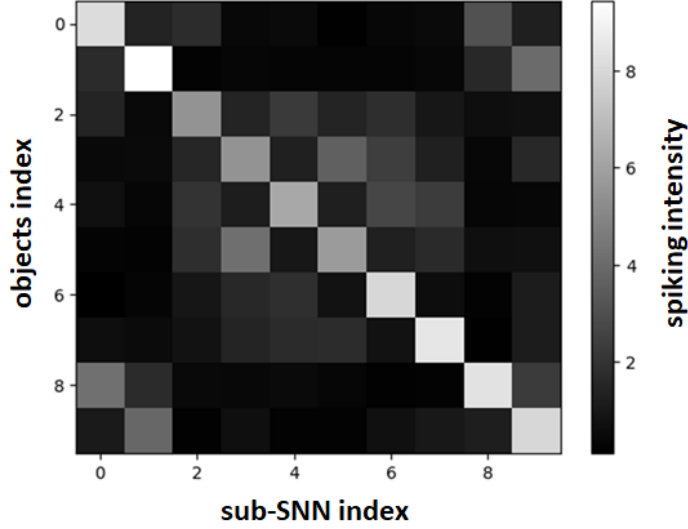


Figure 3.5: This figure shows the spiking intensity of different sub-SNNs (in this example we have 10) when samples from different objects is applied (here we have 10 objects)

the baseline accuracy. In fact, we choose in our experiments a scale factor of 2 to speed up the training process. Other fine-tuned scaling techniques for the width can also be adopted.

For classes containing objects of similar difficulty, this process results in scaling down the sub-SNN width by $1/C$ on average as will be shown in the evaluation section. In addition, the efficiency of a sub-SNN inference can be further improved by splitting it into further subnetworks trained on more specialized data (i.e., dividing the data corresponding to class c into further subsets).

3.2.5 Introducing Dependencies Between the Deepest Features Via Simple Aggregation Function

The previously described structuring and training are performed for each sub-SNN independent from other sub-SNNs. Although this improves the overall computational efficiency, it reduces the overall accuracy. In fact, more dependencies on the low-level features provide the baseline SNN with more information to produce diverse higher-level features. This allows the fully-connected classifiers to amplify the spiking activity for the neurons (features) of the target class and attenuate the activity of other neurons. This aspect is less feasible when training each sub-SNNs; i.e., the narrower layers of the sub-SNNs allow less features to be learned at the early layers.

Fig. 3.5 shows the average spiking activity of the output neuron of every sub-SNN

for objects from different classes. As seen in this figure, some sub-SNNs generate, with a high probability, false positives for some objects. This means that employing a trivial decision-making from the sub-SNNs, e.g., tracking the output of the sub-SNN that fires, does not result in sufficient accuracy.

Aggregating the results of the sub-SNNs. Even though some sub-SNN generates false positives with high probability, these false positives correspond to specific objects (not distributed for all objects). We utilized this consistency in false positives to combine the deepest features (i.e., the output of every sub-SNN) using a simple *aggregation function*. This introduced dependencies at the output retains the overall accuracy of the baseline SNN, while significantly improves the computational efficiency due to the elimination of dependencies at low-level features. To not introduce a computational overhead, a simple *weighted average voting* aggregation function is used. For every final output (the output corresponding to each class), this function takes every output of the specialized sub-SNN, multiplies it by a weight and adds it to the final output. This assigns a weighted dependency to every sub-SNN based on the target output class. The average weighting aggregation function is mathematically summarized as following:

- $y_i = \text{subSNN}_i(x)$; x is the input, y_i is the output of the i th sub-SNN.
- $\vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_c \end{pmatrix}$; $\vec{y}$ = the output of c sub-SNNs.
- $\vec{y}_{\text{weighted}} = \mathbf{W} \cdot \vec{y}$; $\mathbf{W}$ is the weighting matrix and $\vec{y}_{\text{weighted}}$ is the final output vector

The weighting matrix $\mathbf{W}$ is separately trained after training all sub-SNNs; i.e., the training dataset is reused along with the trained sub-SNNs to tune $\mathbf{W}$ to produce the final classification results.

3.2.6 The Efficiency Potential of Using Sub-SNNs

The proposed approach has three potential ways to be utilized for the efficient processing of the SNNs. These potentials can be observed with the use of the computing architectures in Fig. 3.6. The simple architecture can utilize the reduced number of memory and computation operations, which reduces the overall latency and energy; ultra-low power microcontrollers (such as TI MSP430) are typical examples of a simple architecture. The temporal architecture can utilize the reduced reuse distance of the

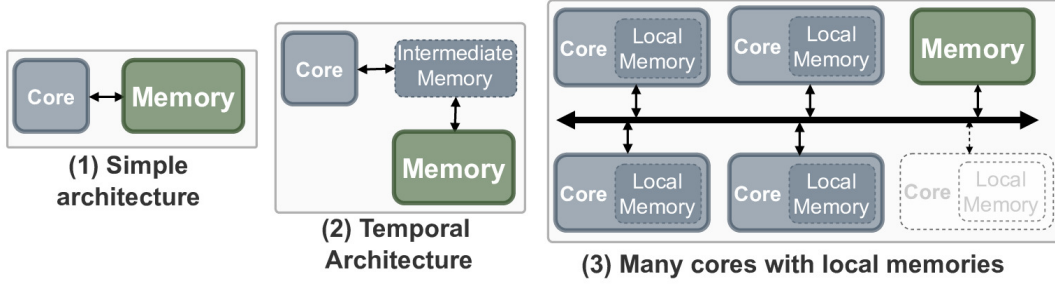


Figure 3.6: Abstraction of different typical computing architectures.

sub-SNNs due to the narrower layer’s width. This allows part of the memory operations to access data in the efficient intermediate memory. Examples of this architecture are the computing systems with on-chip and off-chip memories. Finally, architectures with multiple processing elements can utilize the new level of parallelism the sub-SNNs introduce; i.e., the computations of every sub-SNN are completely independent of other sub-SNNs, allowing each sub-SNN to run on a separate processing element. Examples of spatial architectures include multi-core processors and neuromorphic chips.

3.3 Experiments

In this section, we present experimental results to evaluate the performance of the proposed approach. It is worth mentioning that our energy and latency evaluations were not performed using real hardware platforms as our approach is hardware independent.

3.3.1 Evaluation Methodology

Experimental setup. We used `snnTorch` [25] to integrate the SNNs under PyTorch. We evaluated our proposed approach for image classification problems using MNIST and CIFAR10 datasets and also for speech commands classification using Google speech commands V2 (13 classes were picked). We evaluated our work on 4 baseline architectures: (1) MLP trained on MNIST, (2) VGG9 trained on CIFAR10, (3) VGG9 with skip connections (called here resnet) on CIFAR10, (4) Google speech commands (called here Key Word Spotting - KWS) trained on 3 fully connected layers (3FCLs). In every architecture, homogeneous LIF neurons were used, i.e., all neurons in the same network have the same threshold and decaying factor (time constant). We used our sub-SNN structuring approach to structure the VGG9, RESNET and 3FCLs baseline architectures, while we trivially structured sub-SNNs from the baseline MLP by scal-

3.3. Experiments

Table 3.1: Evaluated baseline SNN architectures

Operation	Latency (ns)	Energy (pJ)
Core operation* [106]	0.5	0.1
SRAM access**[78]	1.5	1.3
DRAM access [43]	60	30

* averaged for add, compare and branch instructions.

** averaged for different memory sizes.

Algorithm 3: Executing a FC layer L in SNN

```

1 for every input neuron  $i$  do
2   if  $S_i^{L-1} = 1$  then // if input neuron spikes
3     for every output neuron  $j$  do
4       update  $v_j^L$  // update mem. potential
5       if  $v_j^L > v_{th}$  then
6          $S_j^L = 1$  // output neuron spikes

```

ing its width by $1/10$. We used direct encoding for all our experiments as it achieves higher accuracy [53]. For the KWS, we used a spectrogram of 90×40 as the input. In addition, for training, we used a maximum of 16 time steps to train all architectures. This number of steps is reduced for all architectures during inference. We trained all networks using BPTT with surrogate gradient function for 250 epochs (the gradient function is approximated as a Heaviside function). For stabilization of the training, we used the threshold-dependent batch normalization (tdBN) proposed in [126]. The Adam optimizer was used with learning rate of $5 \cdot 10^{-5}$. For the heterogenous batching, the batch size of B_1 and B_2 was set to 64 while B_3 was set 256. Finally, the gradients of the B_2 gradients (loss) was scaled by $\frac{1}{20}$.

Figures of merit. We evaluated our proposed approach by comparing the accuracy, inference energy and latency with the baseline SNNs. The inference energy and latency were computed by extracting the number of memory, add, compare, and branch instructions per inference. The number of instructions was estimated from the structure of the SNN and the firing activity (the number of spiking neurons step). Our evaluation considers that the SNN implementation skips ineffectual instructions; in particular, skips

all memory and computing operations that correspond to a non-spiking input. As an example, the execution algorithm of an FC layer in an SNN is shown in Algorithm 3. Updating the neurons of a layer (i.e., v_j^L and S_j^L) starts only if an input neuron spikes (i.e., $S_i^{L-1} = 1$). In other words, all computations and memory accesses corresponding to a non-spiking input are skipped.

Host computing systems. The energy and latency per instruction are analytically estimated based on two simple hardware architectures. The first hardware architecture has a single core and a main memory (first architecture in Fig 3.6). This architecture has only one type of memory operation, which always accesses data from the main memory. The second hardware architecture is a temporal architecture (second architecture in Fig. 3.6), which consists of a simple core, an intermediate memory and a main memory. This architecture is similar to the previous architecture except that it has two different types of memory operations: (1) Inefficient memory operation that accesses the data from the main memory, and (2) efficient memory operation that accesses data from the intermediate memory. The utilization of the intermediate memory, which is proportional to the efficient memory operations, is modeled as a proportional value the SNN’s layer. For example, in Algorithm 3, re-accessing v_0^L at the next spiking input occurs after accessing v_j^L for all j . This also applies to convolutional layers, except that a part of the output neurons (in every feature map) is accessed when an input neuron spikes. Table 3.1 shows the energy and latency models used for evaluation: the main memory was modeled as DRAM, the intermediate memory was modeled as an SRAM, and the other computations were modeled as an average core instruction.

3.3.2 Results

Accuracy. Table 3.2 presents the accuracy results of our experiments. For MNIST-MLP and KWS-3FCLs, the accuracy after aggregating the results of all sub-SNN is not affected compared the baseline architecture. This shows that for simple applications, the accuracy after sub-structuring the baseline architecture is maintained. For CIFAR10-VGG9 and CIFAR10-RESNET, the accuracy is degraded compared to the baseline architectures. The dataset used in these experiments is relatively more complex than the other datasets. Therefore, our results show that the accuracy of our approach degraded with increasing the complexity of the dataset. On the other hand, this degradation in accuracy decrease when using more complex architectures (e.g., our approach has an accuracy of 80% for CIFAR10-RESNET and 73% for CIFAR10-VGG9, while the accuracies of the baseline architectures almost remain constant). Finally, it is worth

3.3. Experiments

Table 3.2: Accuracy (%) on different networks and datasets

Experiment	Accuracy
MNIST MLP baseline	95
MNIST MLP sub-SNNs	98
MNIST MLP sub-SNNs (average)	99
CIFAR10 VGG9 baseline	89
CIFAR10 VGG9 sub-SNNs	73
CIFAR10 VGG9 sub-SNNs (average)	90
CIFAR10 RESNET baseline	91
CIFAR10 RESNET sub-SNNs	80
CIFAR10 RESNET sub-SNNs	92
KWS 3FCLs baseline	62
KWS 3FCLs sub-SNNs	62
KWS 3FCLs sub-SNNs (averaged)	84

mentioning that, as shown in table 3.2, the average accuracy of all sub-SNNs before the aggregation function is close or even higher than the baseline accuracy. This implies that the aggregation function contributes to the degradation of the accuracy in some of the experiments.

Number of spikes per step. Fig. 3.7 shows the number of spikes per step for every layer. In our proposed approach, we calculated the number of steps for two types of execution schemes: (1) when all sub-SNNs are executed in parallel, and (2) when the sub-SNNs are executed sequentially. For parallel execution of sub-SNNs, the number of spikes per step for a specific layer is added for all sub-SNNs, while for sequential execution, the average is taken for every layer. Fig. 3.7(a),(c),(e) and (g) show that the number of spikes per step for the parallel sub-SNNs is on average similar to the baseline SNN; in some layers the number of spikes is higher for parallel sub-SNNs and lower for other layers. This means that the specialized SNNs require computing a similar number of features compared to the baseline SNNs. The sequentially executed sub-SNNs produce much lower number of spikes per step compared to the baseline SNN.

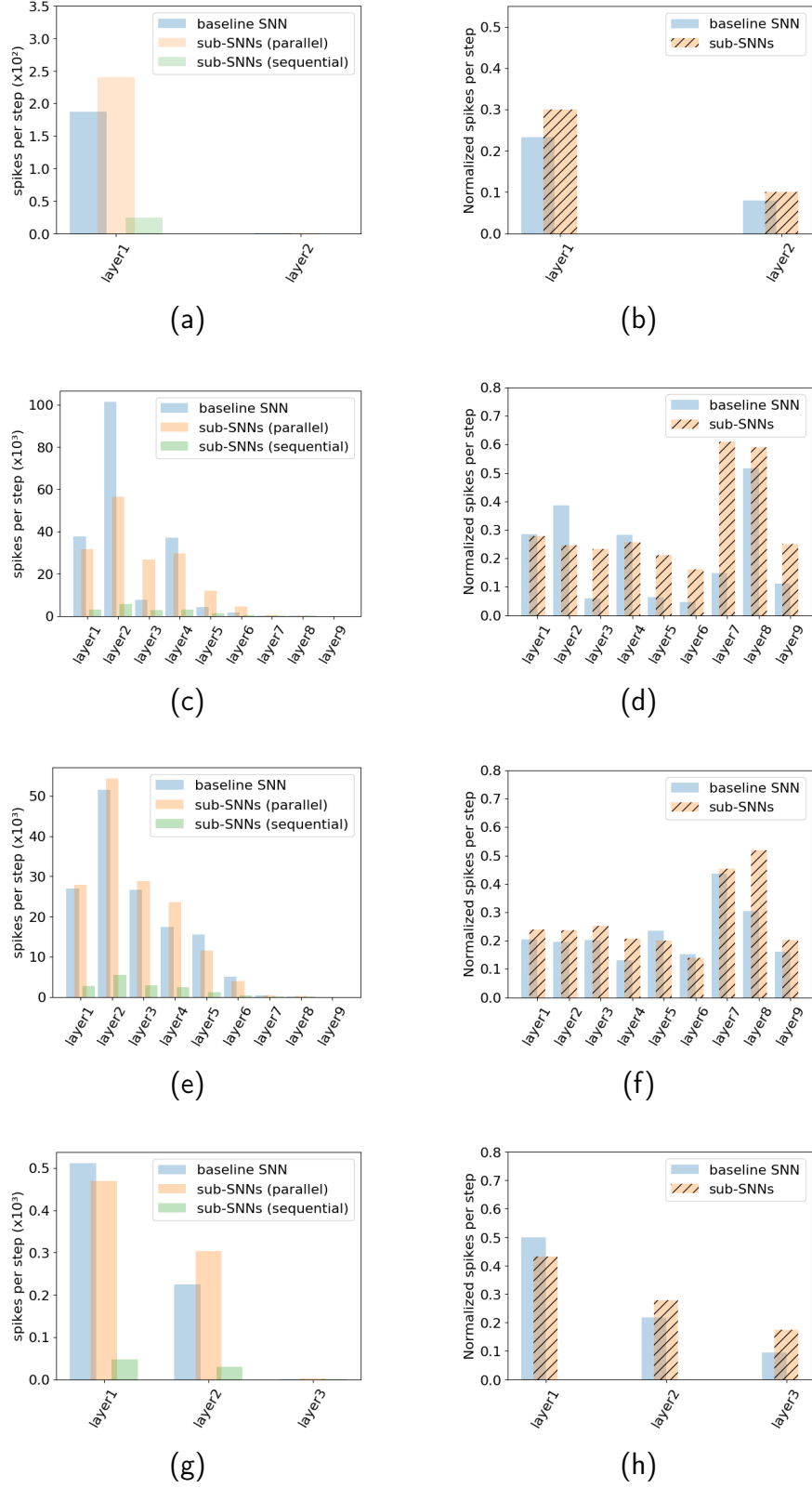


Figure 3.7: Spikes per step: (a) MNIST-MLP, (c) CIFAR10-VGG9, (e) CIFAR10-RESNET, (g) KWS-3FCLs. Normalized spikes per step: (b) MNIST-MLP, (d) CIFAR10-VGG9, (f) CIFAR10-RESNET, (h) KWS-3FCLs.

3.3. Experiments

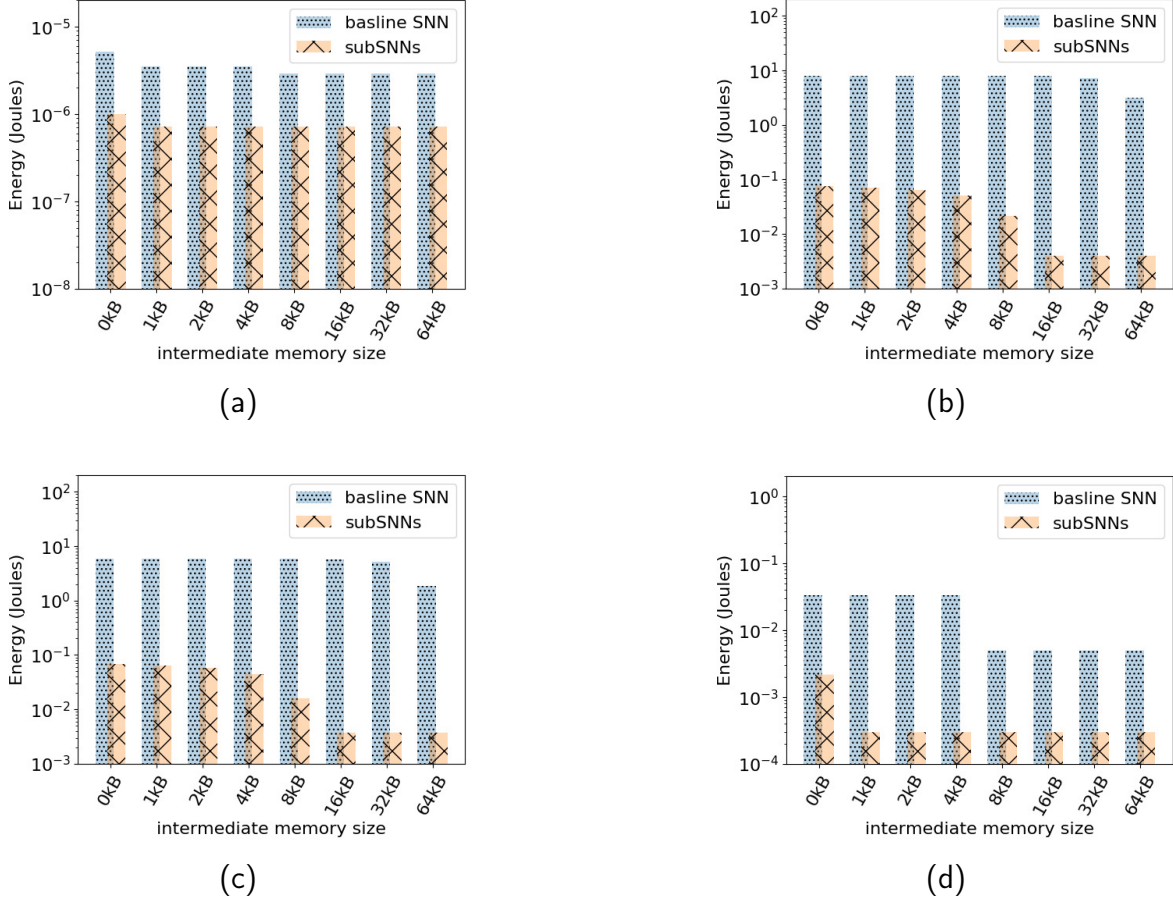
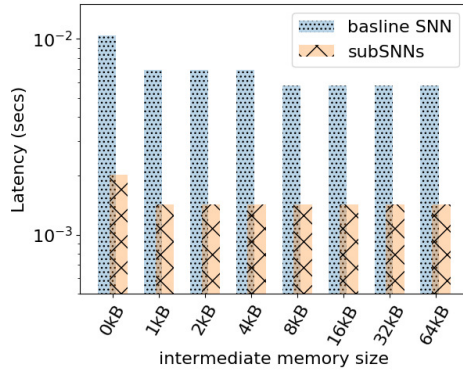


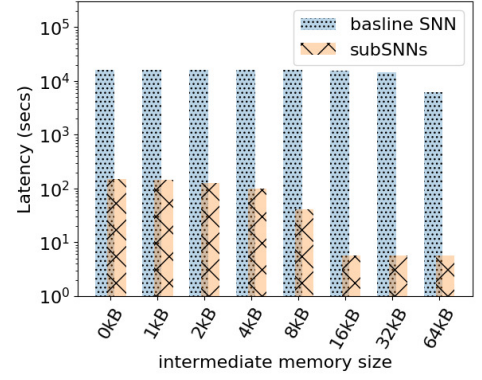
Figure 3.8: Energy: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.

This means that there is much lower number of operations needed per step. Fig. 3.7(b), (d), (f) and (h) shows the percentage of the spiking neurons per step (the number of spikes per step normalized by the layer width) for both the parallelly and sequentially executed sub-SNNs. On average, the proposed approach increases the number of active neurons per layer; this can be seen in most layers in Fig. 3.7(b), (d), (f) and (h). It is worth mentioning here that the number of spikes or the active neurons per layer does not precisely reflect the energy and latency. The energy and latency also depend on the structure of the SNN as will be shown next.

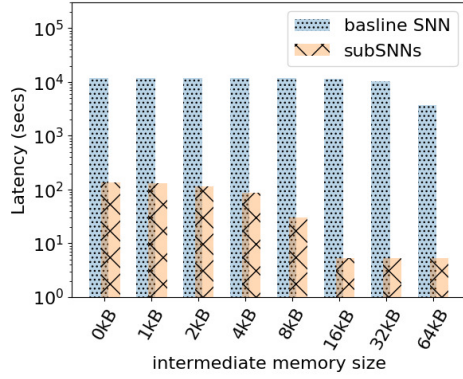
Energy and latency. Fig. 3.8 and Fig. 3.9 show the energy and latency results when executing the baseline SNN and the sub-SNNs on a simple architecture without an intermediate memory and the same architecture with different sizes of intermediate memory. These results are shown in the logarithmic scale to be able to visualize the difference between the baseline SNNs and the sub-SNNs. The results show at least one



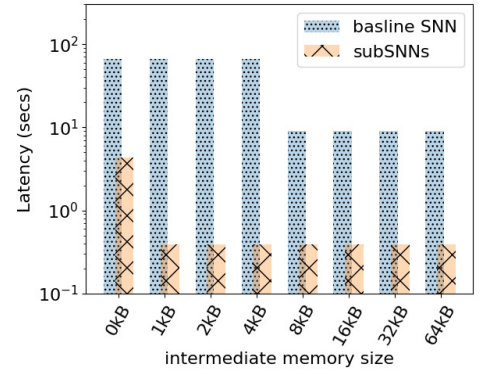
(a)



(b)



(c)



(d)

Figure 3.9: Latency: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.

order of magnitude gain in both energy and latency. The execution on an architecture with no intermediate memory shows the gain of our approach in terms of the number of memory operations. In addition, compared to the baseline SNN, the sub-SNNs make better utilization of an intermediate memory. This is because narrower layers have smaller reuse distances. E.g., a spike in an input layer requires updating all membrane potentials in the output layer. This means that a membrane potential at one layer is reused after updating the states of all membrane potentials in the same layer, making the reuse distance proportional to the layer’s width. Moreover, better utilization of the intermediate memory relaxes the requirements on the memory size; i.e., the runtime memory requirements.

Summary of results. Our evaluation showed that our approach has no impact on the accuracy for a simple application such as (e.g., CIFAR10-VGG9 and CIFAR10-RESNET). The degradation of accuracy is observed to be reduced under more complex architectures (e.g., our approach performed better for resnet architecture compared to VGG9 for the same dataset). Our approach improves the inference energy and latency by more than one order of magnitude in terms. In addition, the narrower widths of sub-SNNs allow them to better utilize an intermediate memory of different sizes compared to the baseline SNN. This allows the proposed approach to relax the runtime memory requirements of the SNNs.

Recall: Q3. What is the quantified gain in terms of processing efficiency when reducing the memory overhead of an SNN?

This chapter presents the quantified gain in processing efficiency when reducing the memory overhead using task specialization. The results show an improvement of at least 10x in terms of energy and latency. This improve is a result of reducing the number of memory operations and improving the utilization of a hierarchical memory.

3.4 Limitations

Scaling application complexity. Our approach considerably reduces the memory overhead of SNNs; at least, for one order of magnitude for both energy and latency. On the other hand, our results show a limitation in approach when scaling the complexity of the application. This urge a need in future work to investigate the applications

that can benefit from the improvements in energy and latency of our approach without sacrificing the accuracy. Another important aspect to investigate is the design of a more complex aggregation function that increases the accuracy without introducing a large computational overhead.

Training burden. Although our proposed approach improves the inference energy and latency, it introduces a burden during training. First, training sub-SNNs requires a re-arrangement of data to create three sets of batches for every sub-SNN. This creates a huge memory and/or latency during the training. Second, the sub-SNNs structuring further makes training inefficient. We have to retrain each sub-SNN for different widths to achieve acceptable accuracy. Therefore, we plan in future work to propose different techniques to train and structure the sub-SNNs to reduce the burden in the training process.

Network pruning. Our approach shares some properties with structured pruning. In particular, removing the dependencies at the low-level features is similar to removing the weights between these features. However, our approach is different than pruning in multiple aspects. First, our approach preserves the total number of features in the baseline SNN. This allows the sub-SNNs to have the same redundancy as the baseline SNN, which allows our approach to be integrated with pruning techniques; we plan in future work to further optimize the sub-SNNs by integrating network compression techniques into our approach. Second, our approach structures the sub-SNNs by consistently reducing the width, which does not only reduce the dependencies but also reduces the reuse distance, which allows the proposed approach to make better utilization of the memory hierarchy; we plan in the future to further specialize the sub-SNNs to reduce their widths even more. Finally, our approach allows us to conditionally execute the sub-SNNs, such that if a sub-SNN detects its object, the execution terminates and the other sub-SNNs are skipped. This allows us to significantly reduce the number of memory and computational operations. We aim to implement conditional execution in future work.

Hardware implementation and intensive evaluations. Our evaluations were based on simplified hardware models. More accurate results can be achieved with an implementation on real hardware. Real hardware can be a resource-constrained microcontroller, a neuromorphic chip, FPGA, etc. We plan in future to test on a variety of hardware resources to have a more realistic idea of the latency and energy gains. Moreover, we also plan to evaluate the proposed approach with other input encoding schemes, datasets, and network architectures.

Utilizing the proposed approach on DNNs. It is worth mentioning that the

proposed approach can also be extended to DNNs. We choose SNNs as they are considered an efficient replacement for DNNs. In addition, SNNs have a larger memory burden than DNNs, i.e., each spiking neuron in any layer requires a variable to store the membrane potential, which cannot be removed from the memory after completing the execution of the layer’s computations. Therefore, our proposal naturally fits the SNNs. On the other hand, we aim to deploy our proposal approach in DNNs and evaluate any possible gain in terms of energy and latency.

The impact of the number of sub-SNNs. With the current version of our work, the number of the sub-SNNs is automatically decided by the number of classes in the image classification application. On the other hand, some image classification applications require a large number of classes. In this case, fine-tuning the specialized sub-SNN for every class might have a large training overhead or wide layers for some complex classes. Therefore, we might need some sub-SNNs to specialize on more detecting and classifying multiple classes with more mutual visual features. We plan to investigate this part in future work.

3.5 Conclusion

We proposed an approach that performs a classification task using multiple specialized SNNs collaboratively. This specialization is utilized to reduce the dependencies at low-level features, which reduces the number of memory operations, increases the efficiency of memory operations, and increases the level of parallelism. This makes the proposed approach more energy efficient and faster than the conventional SNNs. We achieved at least a 10x reduction in energy and latency without affecting the accuracy or the memorization capacity of the SNNs. This shows that replacing dependencies from shallow to deeper features allows us to significantly improve the processing efficiency of SNNs. In future work, we aim to integrate our proposed approach with other approaches that aim at increasing the efficiency of the SNNs, such as network pruning, quantization, sparse-spiking approaches, and efficient data representations. In addition, further investigation of the proposed approach on real hardware is needed.

Chapter 4

Improving the Specialization by Per-Layer Specialization

This chapter presents the second proposed solution of using gated and specialized layers in SNNs. This approach reduces memory overhead while maintaining the accuracy. The per layer specialization dynamically scales the width of the layers in the network, reuses some specialized experts by different classes, and eliminates the training burden that existed in the previous task specialization approach. Our results show an improvement in inference processing efficiency on real general-purpose hardware of up to 3x while maintaining state-of-the-art accuracy. The content of this chapter was published in the following paper: Muath Abu Lebdeh et al. “Per layer specialization for memory-efficient spiking neural networks,” Proceedings of International Conference on Neuromorphic Systems (ICONS), pages 169–176, 2024.

4.1 Introduction

The previous chapter showed the effectiveness of task specialization in reducing the memory overhead of SNNs. Task specialization approach divides the classification task into multiple classification sub-tasks, each of which is trained on a narrow SNN, where these narrow SNNs collaboratively produce the final inference result. Although this significantly reduces the memory overhead of the SNN, it still presents non-negligible drawbacks and challenges.

First, the task specialization approach scales the width of all layers of a specialized SNN by the same factor, which results in a high degradation of accuracy for some applications and architectures. Second, it requires redistributing the training dataset

for the heterogeneous batching and gradient accumulation, which is a huge training burden that slows down the training process. Third, the task specialization approach is not guaranteed to be effective in real word applications, such that the divided sub-tasks cannot be guaranteed to have unique representations (i.e., separable representations). In practice, designing the sub-tasks requires a human expert in the semantics of the image classes to guarantee mutually exclusive representations.

To solve these challenges, we propose the per layer task specialization in the SNNs. In this approach, every layer in a single SNN model is dynamically specialized rather than using multiple independent narrow SNNs. This allows for an input-dependent narrow dataflow, such that every input activates a small and narrow part of the model in every time step. We implement the per layer using the sparsely gated models, where each layer is divided into independent and narrow experts. This approach reduces the overall memory overhead and computational complexity, leading to gains in the processing efficiency of SNNs. For example, the proposed approach achieved a 3x faster inference for a customResNet-9 trained on cifar10 dataset, while maintaining the state-of-the-art accuracy of similar architectures. In addition, the proposed approach does not introduce any training burden compared to the previous work in task specialization; during training, a trainable gating function automatically specializes the experts in every layer. Finally, the proposed approach inherently allows for computation skipping, which has important effects on the memory requirements of the SNNs. In summary, the contributions of this chapter are:

- **A systematic and trainable specialization of tasks in SNNs:** This reduces the training burden as well as the need for human experts to design the specialized image classification tasks
- **A per layer specialization of the image classification task:** This specializes the SNN model in a fine-grained way, which results in a higher accuracy compared to the previous work in SNN task specialization.
- **Introducing the concept of conditional computation in SNNs:** This improves the processing efficiency of the SNNs by skipping the majority of computation and memory operations as well as increasing the utilization of a hierarchical memory system.

4.2 Proposed Solution: Per Layer Specialization

This section presents the proposed task specialization per layer. It presents some preliminary observations in SNN, the proposed gated SNN architecture, and the used auxiliary losses for load balancing.

4.2.1 Preliminary Observations: Unique Representations of Semantically Different Classes at Deeper Layers

The main concept behind task specialization is that semantically different classes tend to have unique and hence separable representations within the SNN. A unique representation in a layer can be seen as a unique spiking pattern, i.e., specific neurons spike in every layer per class. Therefore, if an image classification task is divided into sub-tasks that are semantically different, we expect their representation to be unique in every layer, hence we may train every sub-task on a narrow specialized SNN. In [56], all layers of a specialized SNN are scaled by the same factor; i.e., the width of every layer is reduced by the same factor. However, we discovered that every semantically different class (sub-task) does not have unique (separable) representations in every layer. Semantically different classes tend to have unique representations at deeper layers, i.e., when the layer is deeper, the features become more unique and distanced. In other words, in the work of task specialization, all layers are scaled by the same factor, while deep layers are task-dependent but early layers are not task-dependent. To observe this property, we adopt the visualization in [72]: at a specific layer in the SNN, the features in this layer are vectorized and then projected into a 2D plane for visualization.

Fig. 4.1 shows the projection of features in CustomResNet-9 trained on the CIFAR10 dataset. Each figure shows the projection of the features of three classes. Fig. 4.1a shows the projection of features of three semantically different classes; ‘airplane’, ‘automobile’, and ‘cat’. In the leftmost figure (projections in the second layer), the projections are less separable compared with the projections in the rightmost figure (projections of the feature after the deepest layer). This is more obvious when visualizing the projections of two semantically similar classes (‘cat’ and ‘dog’) and one semantically different class (‘airplane’) as shown in Fig. 4.1b. These figures show that in earlier layers, the representations of features belonging to different classes are less unique than the representations of features at deeper layers. Therefore, specialization at deeper layers is easier and more applicable than the specialization in earlier layers. Thus, scaling all layers by the same factor results in a non-negligible drop in accuracy.

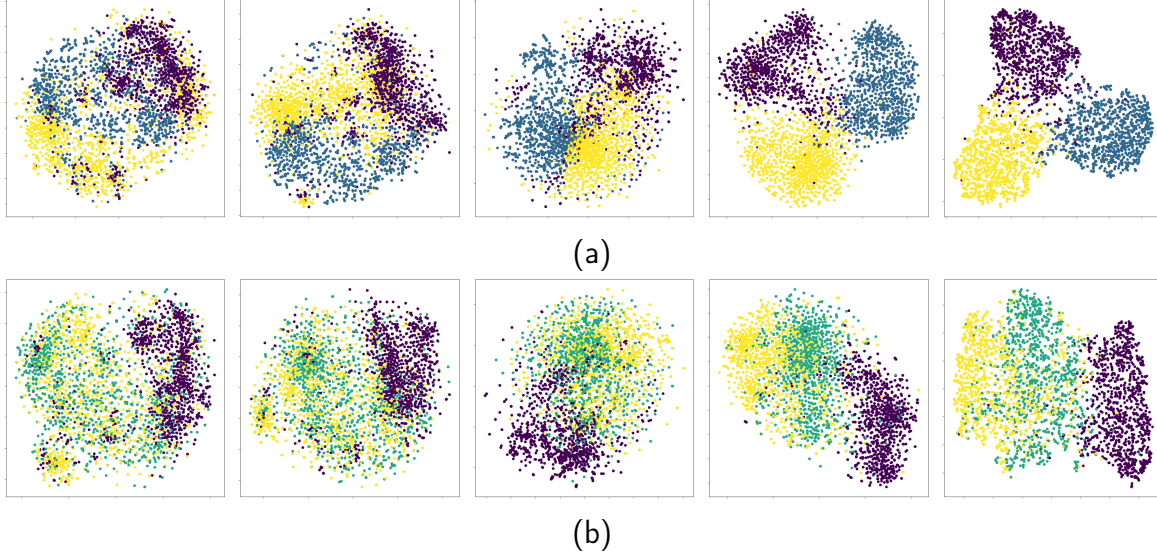


Figure 4.1: Projection of features at different layers into 2D plane for CustomResNet-9 trained on CIFAR10 dataset. (a) shows the projections of features for three semantically different classes: ‘airplane’ (magenta), ‘automobile’ (blue), and ‘cat’ (yellow). (b) shows the projections of features for 2 semantically similar classes, ‘cat’ (yellow) and ‘dog’ (green), and one semantically different class, ‘airplane’ (magenta). The depth of the layer increases from left to right (i.e., the rightmost figure shows the projection of the deepest features).

Based on these observations, we decided to specialize the layers themselves within the SNN as shown in Fig. 4.2. Every layer is composed of independent narrow experts (at maximum each expert has two layers). The level of specialization in every subsequent layer is increased by some factor (e.g., the first layer has 1 expert, the second has 2 experts, the third has 4 experts, etc). With every specialization level, the width of the expert is narrowed by some factor (e.g., the width of each expert in the second layer is scaled by $1/2$, the expert of the third layer is scaled by $1/4$, etc).

There are three major differences from the earlier approach in task specialization [56]. First, in this work, the early layers are shared and used to extract the features from all sub-tasks, while in the previous work, every layer is trained for a specific sub-task. Second, in this work, the specialized architecture is directly trained on the whole dataset without redistributing it for specialization. This eliminates the training burden that existed in the previous work. Third, this work uses a simple gating function in every layer to selectively activate one expert. This allows for skipping most of the computations and memory operations in deeper layers. In addition, this skipping reduces the memory requirements of the SNN (especially in the direct encoding scheme), where only the membrane potentials of the selected experts are maintained in the

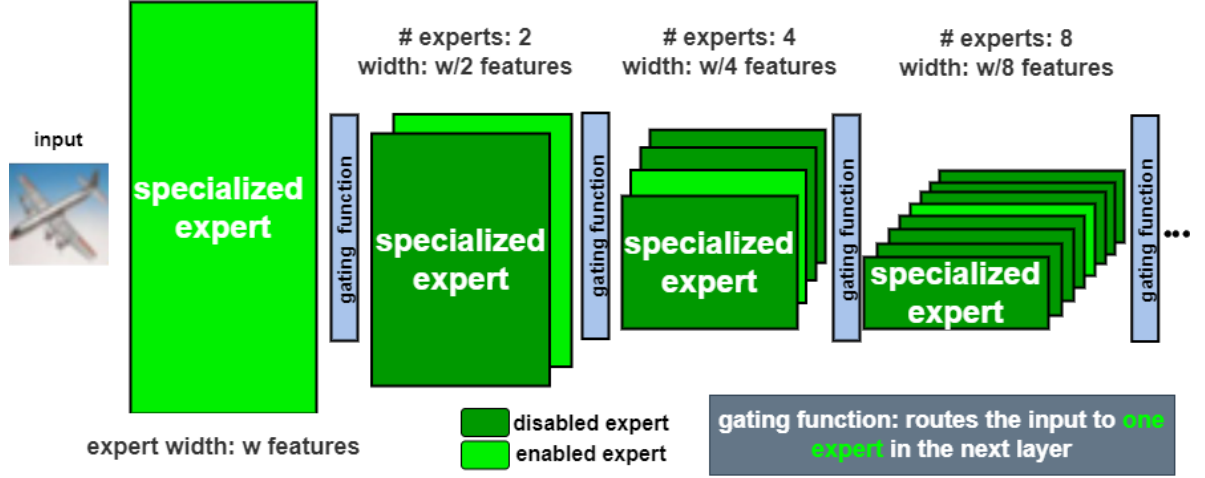


Figure 4.2: The proposed architecture for task specialization per layer. The gating function routes the input features to only one of the specialized experts.

memory. More details of the proposed task specialization per layer are presented in the next subsection.

4.2.2 Gated Specialized Models

The proposed task specialization per layer combines two important design choices:

1. *Sharing the early layers*: As specialization in early layers results in a non-negligible accuracy drop, sharing (non-specializing) is emphasized among all sub-tasks. This avoids designing wide early layers for every sub-task, and hence eliminates unnecessary redundancy; note that early layers are used to extract general features, such as edges, blobs, and other simple geometrical features in image applications.
2. *Gating the specialized experts*: Some deeper specialized experts can be used by different sub-tasks. A gating function (similar to the gating function used in the sparsely gated mixture of experts in [89]) selects one specialized expert, i.e., the gating function routes the input features of every layer to only one specialized expert. This allows for: (1) reducing the number of the features produced by every layer (features are produced by one narrow expert), and hence reduce the overall memory overhead, (2) re-using the experts for more than one sub-tasks, and hence reducing the size of the model, (3) alleviating the training burden of the conventional task specialization approach as the gating function is trainable, and hence automatically designs the sub-tasks and automatically assigns these sub-tasks to the suitable experts.

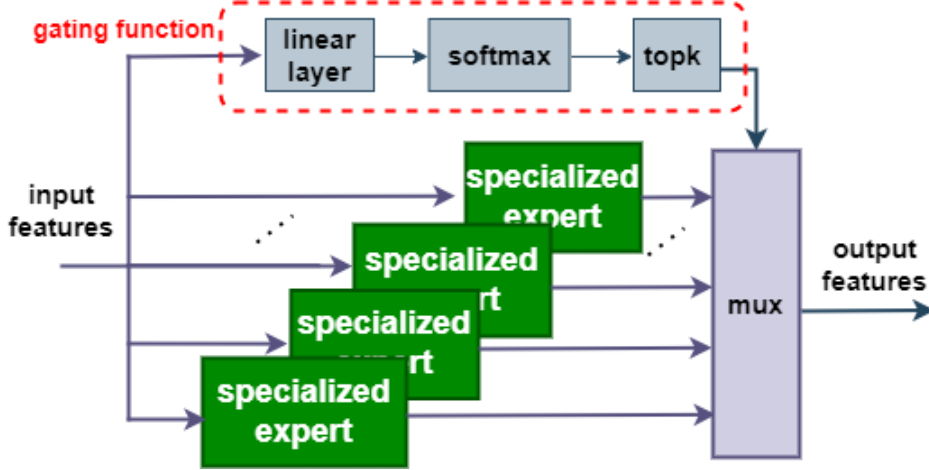


Figure 4.3: The gated layer used during training. One expert is selected at every time step.

The idea of gating is inspired by the idea of sparsely gating a layer of experts proposed in [89]. Our work differs from the previous work in two important aspects. First, rather than using transformers, our work uses only spiking neurons, except for the gating function, which (as shown in Fig. 4.3) is composed of a linear layer, followed by a softmax function for normalization, and a top1 function to select one of the experts. Second, rather than applying the input image as a sequence of 2D patches, we apply the whole image as the input in all time steps (direct encoding). This allows the same features to be extracted in every time step and, hence, guaranteeing that almost the same experts are activated in all time steps. Therefore, this reduces the number of stored membrane potentials in memory and increases the utilization of a hierarchical memory system.

4.2.3 Balanced Usage of Experts

With only using the classification loss, the gating function converges during training to select only one expert for all classes. This unbalanced use of experts during the inference leads to reduced accuracy. We adopt the use of the auxiliary losses proposed in [89]. A balanced use of experts in every layer is encouraged using a regularization term added to the classification loss as defined below:

$$\mathcal{L}_{total} = \mathcal{L}_{classification} + \mathcal{L}_{auxiliary} \quad (4.1)$$

$$\mathcal{L}_{auxiliary} = \mathcal{L}_{Imp} + \mathcal{L}_{Load} \quad (4.2)$$

The importance loss $\mathcal{L}_{Imp}$ is used to balance the distribution of softmax outputs in the gating function, while the load loss $\mathcal{L}_{Load}$ is used to balance the assignments of the topk function.

The importance loss in every gating function relies on the importance score which is computed as following:

$$\text{Imp}_i(\mathbf{X}) := \sum_{x \in \mathbf{X}} \text{softmax}(Wx)_i, \quad (4.3)$$

where i is the expert index, X is the batch of input features in every layer, W is the weights matrix of the linear layer in the gating function. The importance loss of every layer is computed from the standard deviation and the mean of all experts' importance scores as follows:

$$\mathcal{L}_{Imp}^l = \left(\frac{\text{std}(\text{Imp}(X))}{\text{mean}(\text{Imp}(X))} \right)^2 \quad (4.4)$$

Finally, the total importance loss is computed by adding the importance losses of all layers.

The load loss is used to guarantee a balanced assignment by the topk function; i.e., balancing the output of the softmax balances the distribution of its output but not necessarily the assignments. A differentiable approximation of the top-k function is computed as the CDF of a normal distribution:

$$\begin{aligned} P_i(x) &= P(\epsilon < (Wx) - \text{threshold}_k(x))_i \\ &= \Phi((Wx) - \text{threshold}_k(x))_i, \end{aligned} \quad (4.5)$$

where ϵ is a noise sampled from a standard normal distribution (of zero means and $1/4E$ standard deviation; E is the number of experts in the layer), Φ is the CDF of a standard normal distribution, $\text{threshold}_k(Wx)$ is the maximum k element of Wx , and i is the expert index.

The load loss for every layer is computed from the standard deviation and mean of all experts' probability scores as follows:

$$\mathcal{L}_{Load}^l = \left(\frac{\text{std}(P(X))}{\text{mean}(P(X))} \right)^2 \quad (4.6)$$

The total load loss is computed by adding the load losses of all layers.

4.3 Experiments and Discussion

This section presents the evaluation of our approach and a discussion of the results.

4.3.1 Evaluation Methods

Experimental setups. We evaluated our results using fashion-mnist and CIFAR10 datasets. These datasets were trained on three different architectural variations of the proposed approach. (1) *Dynamic specialization and hardgating*: An architecture with dynamic scaling of layers and a hardgating function; where layers are specialized with different levels and one expert is activated in every layer. (2) *Dynamic specialization and softgating*: An architecture with dynamic scaling of layers and softgating function; where layers are also specialized with different levels, while all experts are executed and their outputs are weighted and added (gating function without a topk unit). (3) *Constant specialization and softgating*: An architecture with constant scaling of layers and softgating function; this is similar to the previous architecture but with a constant specialization level [56].

We used 4 steps, the LIF neuron model and BPTT to train all SNN architectures. We used 400 epochs with a cosine annealing scheduler. We did not use any normalization through time in all trained architectures. This allowed us to speed up our experiments during training.

Evaluation metrics. We evaluated and compared our proposed approach with the state-of-the-art in terms of accuracy and processing efficiency. As a metric of the processing efficiency, we measured the inference latency of the SNN model in PyTorch on a general-purpose CPU (Intel Xeon Gold 6252N); as computations are executed sequentially, this metric can be a practical estimate of both the number of operations and the utilization of hierarchical memory. The inference latency was compared with the state-of-the-art by re-implementing a baseline SNN model from the literature. We chose CustomResNet-9 [56] as it consists of diverse architectural concepts such as CONV layers, FC layers and skip-connections. We also chose 6FC layers model [82] as a baseline for simple SNN models. Then, we applied our proposed specialization approach to these baseline architectures. Beside the accuracy and inference latency, we also presented some results on the memory efficiency of the proposed approach as well as the gating pattern of the activated experts.

Software implementation. We implemented the algorithm of the SNN in a layer-wise manner. In this implementation, each layer is executed for all steps, such that the

Table 4.1: Accuracy comparison on different architectures and datasets

Architecture	Dataset	Accuracy
CustomResNet-9 with task specialization [56]	CIFAR10	80
VGG-16 [34]	CIFAR10	87
CustomResNet-9 [56]	CIFAR10	90
ResNet-12 [55]	CIFAR10	90
CIFARNet [34]	CIFAR10	92
ResNet-19 [34]	CIFAR10	95
CustomResNet-9 with constant spec. (of 4) with softgating	CIFAR10	80
CustomResNet-9 with dynamic spec. with softgating	CIFAR10	89
CustomResNet-9 with dynamic spec. with hardgating	CIFAR10	88
6FC layers [82]	fashion-mnist	82
CSNN [82]	fashion-mnist	86
CSNN [82]	fashion-mnist	86
6FC layers with constant spec. (of 6) with softgating	fashion-mnist	82
6FC layers with dynamic spec. with softgating	fashion-mnist	81
6FC layers with dynamic spec. with hardgating	fashion-mnist	80

input in all time steps is applied and a tensor of the output in all steps is produced. In this implementation, the membrane potentials are only saved for the layer being executed. Once all outputs are produced for all steps, the membrane potentials of the layer being executed are removed from memory, and the output becomes the input of the next layer. This implementation is useful for deep architectures operating with a small number of steps.

4.3.2 Results and Discussions

Accuracy

Table 4.1 compares the accuracy of the proposed architectures to SNN architectures from the literature.

Accuracy of CustomResNet-9 trained on CIFAR10. The proposed approach under dynamic scaling with hardgating and softgating architectures have almost the

same accuracy; 88% with hardgating and 89% with softgating. This shows that many computations at deeper layers can be skipped (by hardgating) without affecting the accuracy.

The proposed approach under constant scaling (of 4) and softgating has an accuracy of 80%. This implies that for this architecture and dataset, specialization at early layers leads to a reduction in accuracy. In other words, early layers produce features that are not very dependent on the semantics of the input compared to deeper layers.

In comparison with similar SNN architectures from the literature, the proposed approach under dynamic scaling has close accuracy to the SNN architectures of similar sizes such as CustomResNet-9 [56] (90%), ResNet-12 [55] (90%) and VGG-16 [34] (87%). This shows that the concept of gating and computation skipping does not negatively affect the accuracy.

The proposed architecture under constant scaling has a reduced accuracy compared to the SNN architectures in the literature, but a similar accuracy to the CustomResNet-9 architecture with earlier work in task specialization [56] (80% for both architectures). The proposed approach under constant scaling is architecturally similar to the earlier approach in task specialization, except that a gating function is used in the proposed architecture. This implies that the adaptive weighting of experts' output in every time step has a negligible impact on the accuracy.

Accuracy of 6FC layers trained on fashion-mnist. The proposed approach with all tested architectural variations has a similar accuracy (around 81%). This shows that for some architectures, the type of scaling has a negligible impact on accuracy. Similar results were reported in the earlier work in task specialization [56]: Two simple two fully connected architectures were trained on mnist and Google speech commands, and their accuracy results do not degrade when task specialization is applied.

In comparison to SNN architectures from the literature, the proposed approach has a similar accuracy to the accuracy of 6 FC layers [82] architecture (82%), which has a similar architecture size to the tested proposed architectures.

Summary of accuracy results:

- The proposed approach with hardgating function produces similar accuracy to the same architecture with softgating function. This shows that computation skipping (including skipping the corresponding memory operations) has a negligible impact on accuracy.
- Dynamic scaling of the SNN layers improves the accuracy for relatively wide and deep convolutional architectures, while does not have an impact on simple fully

connected architectures.

- Constant scaling with the direct encoding of the inputs produces similar results in accuracy to the earlier work in task specialization [56]. This emphasizes that the accuracy degradation in the previous work in task specialization was highly a consequence of the constant specialization in all SNN layers.

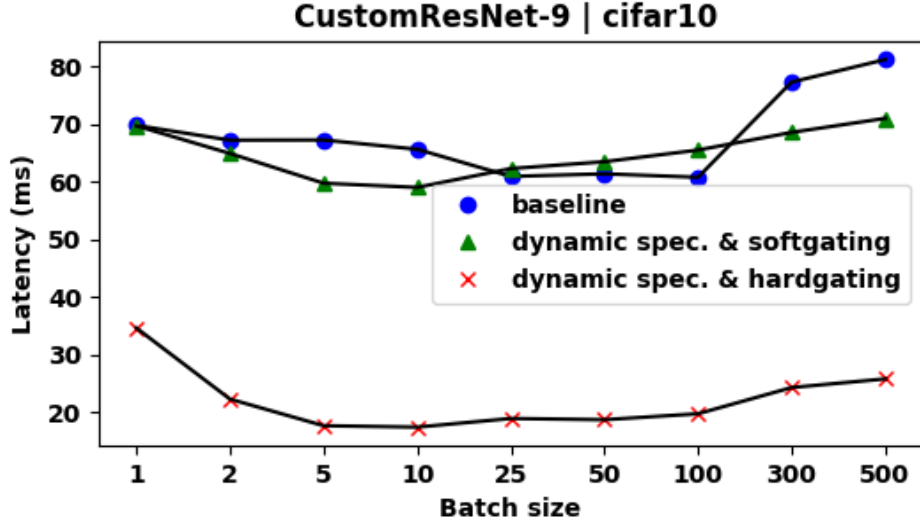
Inference Efficiency

Fig. 4.4 shows the inference latency of running different SNN architectures on Intel(R) Xeon(R) Gold 6252N CPU. The latency here represents the average latency per sample when executing the architecture at different batch sizes. The whole batch is executed per layer, such that every layer is processed for all samples in the batch. The batch sizes in our experiments were tuned from 1 to 500 as shown in Fig. 4.4.

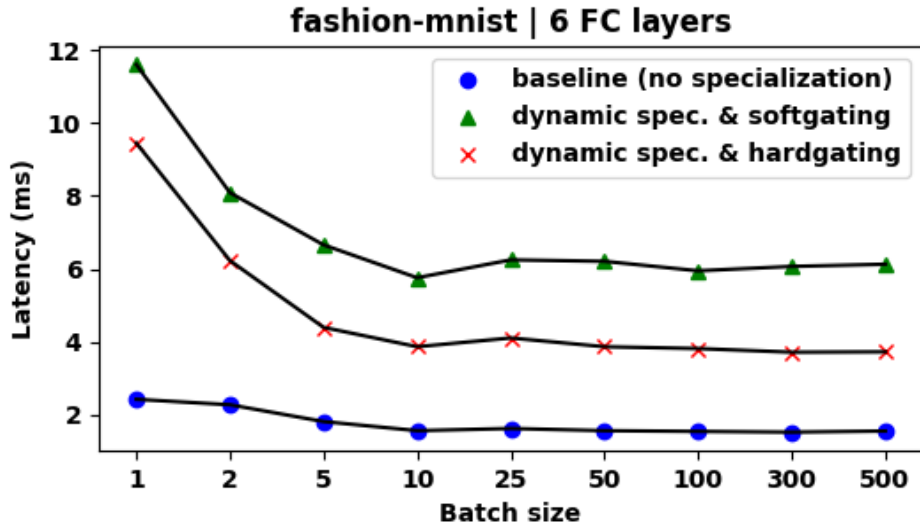
Efficiency of CustomResNet-9 trained on CIFAR10. The hardgating architecture is more efficient than the softgating and the baseline architectures. This efficiency gain in the hardgating is a result of skipping computations and memory operations. The softgating and the baseline architectures have close efficiency at lower batch sizes. The computations required for both approaches are similar except that, in the softgating architecture, there is an overhead for the gating function. In terms of memory overhead, with small batch sizes, the size difference of the intermediate results for both softgating architecture and the baseline architecture is negligible. This explains the close latencies of both architectures with small batch sizes; note that the latency of the baseline architecture is even better in some cases because of the additional computations introduced by the gating function in the softgating architecture.

At higher batch sizes, the softgating architecture starts to be more efficient and the gap increases with larger batch sizes. With larger batch sizes, the impact of memory in the intermediate layers becomes critical. In softgating architecture, the results of the smaller feature maps allow them to utilize better the memory hierarchy of the hardware, and hence it outperforms the baseline architecture. This utilization is also higher for the hardgating architecture. It is worth mentioning that the variation in efficiency because of increasing batch size is similar to increasing width of the network architecture or reducing size of the hardware memory.

In all architectures, the latency starts high, then starts to decrease, then starts to increase again at higher batch sizes. With smaller batches, the overhead of loading the model parameters is high relative to the size of the input and output of every layer.



(a)



(b)

Figure 4.4: Inference latency per sample with different input batch sizes. (a) Shows the latency for architectures trained on CIFAR10 dataset, while (b) shows the latency for architectures trained on fashion-mnist dataset.

In contrast, a large batch size leads to inefficient utilization of the memory as data are shuttled more frequently between different memory levels. A perfect fit of the batch sizes, which depends on the hardware and the network architecture, is between these two extreme cases.

Inference efficiency of 6FC layers trained on fashion-mnist. The baseline architecture is better than the softgating and hardgating architectures. In these relatively small architectures (compared to the hardware memory), the difference between the number of features in every layer is considered negligible; i.e., every layer has a maximum of 300 neurons. Therefore, the compute and branch operations become the dominant operations in latency, in which the gating functions of the gated architectures become critical.

The hardgating architecture is faster than the softgating architecture. This is because of the skipping of computations and memory operations. In all architectures, the latency starts high with small batch sizes, then it starts to decrease when increasing the batch size up to a point where the latency saturates. This is different than the CustomResNet-9 architecture, where in 6 FC layers architecture, all features in every layer fit in the hardware memory.

Summary of inference efficiency results:

- The efficiency of the proposed approach depends on the complexity of the SNN architecture, the availability of the hardware resources, and the size of the input batch.
- The benefit of the proposed approach appears when at least one of the following conditions is met: the SNN architecture is deep and wide, the hardware memory resources are small, or the batch size of the input data is large.
- When none of these conditions are met, the impact of the gating function in the proposed approach becomes critical.

Memory efficiency and gating pattern

Fig. 4.5 presents the memory required for the memory potential variables in a single step for (1) a non-specialized CustomResNet-9 architecture (baseline) and for (2) the same architecture with specialized layers (with dynamic scaling and hardgating function). In the first layer, both architectures use the same amount of memory for the membrane potential variables. This is expected because the specialized architecture uses only one

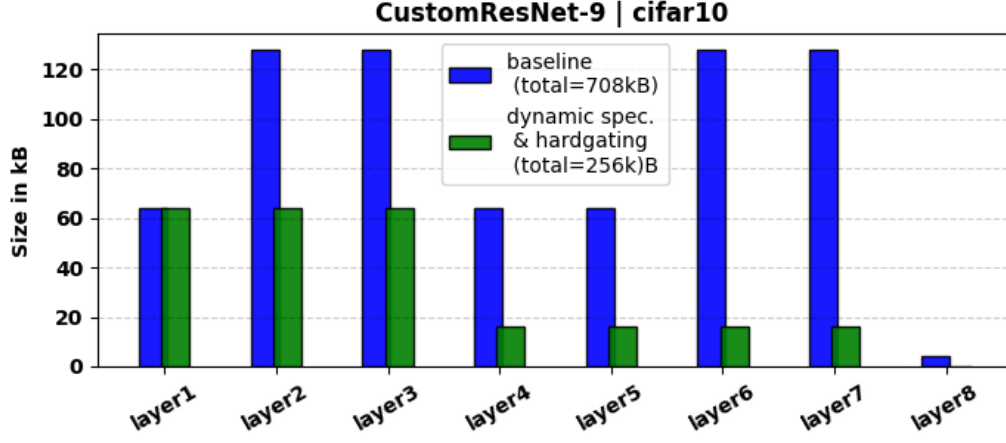


Figure 4.5: The membrane potentials memory required in every time step.

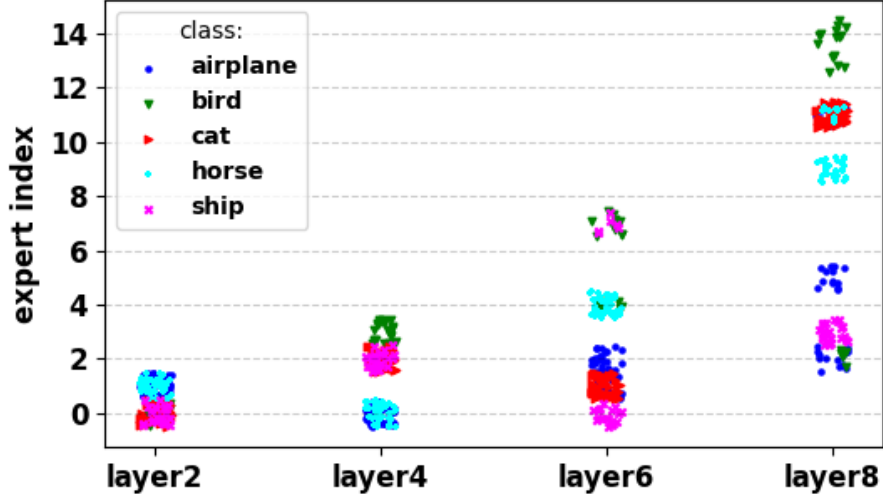


Figure 4.6: The activated experts in every layer for samples from different classes in CustomResNet-9 architecture trained on CIFAR10.

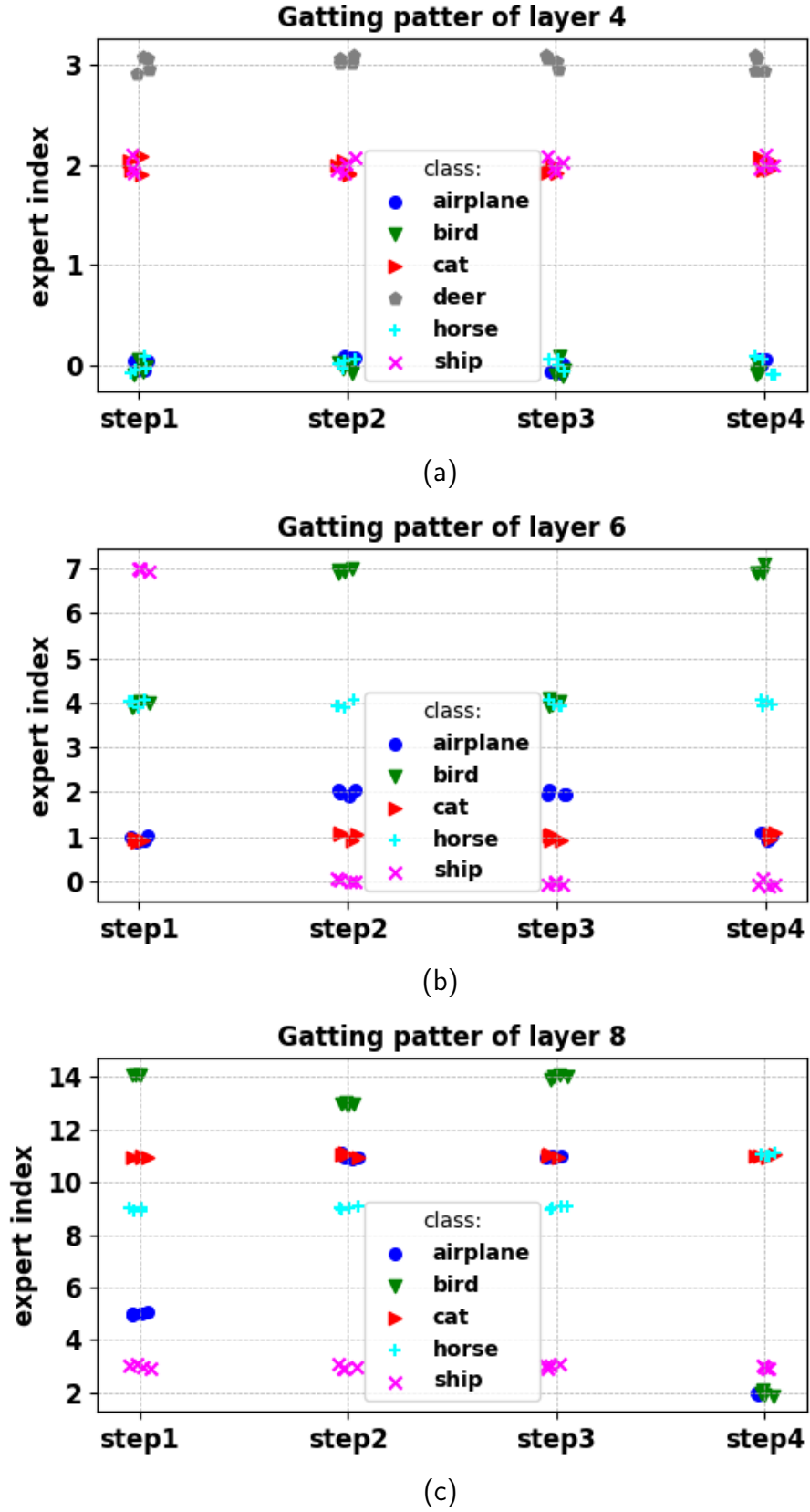


Figure 4.7: The activated experts in every step for samples from different classes in CustomResNet-9 trained in CIFAR10. The results are shown for (a) layer 4, (b) layer 6 and (c) layer 8.

expert of the same size as the first layer in the non-specialized baseline architecture. The difference between the memory required by the two architectures increases with depth. For example, the second layer in the proposed architecture uses almost half of the memory required by the baseline architecture. This difference increases in layer 6, such that the proposed architecture uses 1/8 of the membrane potential memory of the baseline architecture. This increasing difference is due to the increased level of specialization in deeper layers in the proposed architecture. Finally, Fig. 4.5 shows that the proposed architecture requires in every time step less than one-third of the total membrane potential memory required by the baseline architecture. This means that the proposed architecture requires almost one-third of the memory required by the baseline architecture when executed by maintaining all membrane potentials in memory. Note that this is under the assumption that the same expert is activated in every time step.

Fig. 4.6 presents a scattered plot for the activated expert in different layers for different classes in CustomResNet-9 architecture trained on CIFAR10. In every layer, some experts are used by samples from different classes. This can be especially seen in the early layers which have a smaller number of experts. In deeper layers, such as layer 8, the activated experts become more dependent on the class of the input.

Fig. 4.7 shows the activated experts in every step for different layers for samples from different classes. In the early layers, almost the same expert is activated in every step for the same sample. This implies that the required memory for the membrane potentials is equal to the membrane potentials' memory required by a single expert. In deeper layers, such as layer 8, every sample may activate more than one expert in different time steps. However, the results show that the number of activated experts during all time steps does not exceed two experts. This means that the deeper layers require on average 1-3x of the memory requirement of a single expert. Therefore, the proposed approach allows for saving in memory required for membrane potentials proportional to the number of experts; i.e., the higher the specialization in the layer, the higher the savings in the memory.

Summary of memory efficiency and gating pattern results:

- The results show around 3x saving in the memory required by the membrane potentials in comparison with a non-specialized SNN of the same architecture.
- The activated experts in deeper layers are more dependent on the class of the input data.
- The gating pattern (the activated expert) is fixed in almost every step for the

same input. As a result, the saving in the memory is linearly proportional to the number of experts.

- The results implicitly show that the proposed approach could benefit better from the data locality in a hierarchical memory as almost the same expert is activated in every time step.

Recall: Q3. What is the quantified gain in terms of processing efficiency when reducing the memory overhead of an SNN?

The per layer specialization improves the runtime memory requirements by 3x. It also improves the processing efficiency in terms of inference latency (on CPU) by around 3x.

4.4 Conclusion

We proposed in this chapter the specialization of tasks per layer in SNNs. This approach combines concepts from the sparsely gated mixture of experts and task specialization to improve the processing efficiency of SNNs. The proposed approach is capable of improving the processing efficiency of SNNs while maintaining accuracy without introducing a training burden compared to the previous work in SNN task specialization. Our results showed that the proposed approach has a similar accuracy to similar state-of-the-art architectures. In addition, our approach showed improvement in processing efficiency of 3x for CustomResNet-9 architecture. Our results also showed that the overhead of the gating function increases when the size of the intermediate features are small compared to the size of the hardware memory.

Chapter 5

Utilizing Spiking Neurons of Higher Dynamical Complexity

This chapter introduces the approach of utilizing the spiking neuron’s dynamical properties for efficient feature extraction. Although this work focuses on the neuron level, it will be used in the next chapter to design narrow SNNs. This chapter starts with presenting the potentials of sequential processing in terms of memory overhead (Q4), then presents the challenges of sequential processing. This chapter shows the limitation of the leaky integrate-and-fire (LIF) neuron model. The LIF neuron model is commonly used in the literature of SNNs due to its computational simplicity. However, this comes at the expense of omitting the complex temporal dynamics of the spiking neuron, which limits it from extracting complex temporal features. In this chapter, we exploit the temporal dynamics of the biologically plausible spiking neurons for efficiently process static images. We also present some results that show the capability of a single spiking neuron in processing digital signals sequentially. The content of this chapter is included in the following paper: Muath Abu Lebdeh et al. “Memory-Efficient Spatiotemporal Convolutional Spiking Neural Networks for Vision Applications,” paper under submission to the IEEE Transactions in Emerging Topics in Computational Intelligence.

5.1 Introduction

The previous chapters showed the importance of specialization in reducing the number of feature maps in convolutional SNNs. This results in reduced memory overhead and hence improved processing efficiency. This chapter introduces a new direction that utilizes the intrinsic properties of the SNNs to reduce the memory size of the

5.2. Proposed Solution: Efficient Processing Utilizing the Temporal Dynamics of the Spiking Neuron

intermediate feature maps, and hence reducing the overall memory overhead. In this approach, the input image is divided into 2D patches that are applied sequentially as tokens to the SNN. The spatial structure of the SNN extracts the local features in every token, while the temporal dynamics of the spiking neuron extracts the global context (global relationships) between different tokens. This results in reducing the size of every feature map in intermediate layers, and hence reducing the memory overhead.

On the other hand, it is challenging to use the LIF neuron model to extract complex global contexts. In this chapter, we propose the use of more complex and biologically plausible spiking neurons to allow the extraction of complex temporal features. This paves the way to introduce SNN architectures (Q5) that allow for narrow dataflows and hence a reduced memory overhead. The contributions of this chapter are:

- Exploiting the temporal dynamics of biologically plausible spiking neurons in sequential processing of static images.
- Presenting empirical results that show the capability of a single spiking neuron in sequentially processing digital signals.

5.2 Proposed Solution: Efficient Processing Utilizing the Temporal Dynamics of the Spiking Neuron

This section presents first the challenge of sequentially extracting features from images. Then, it presents the limitations of the LIF neuron dynamics to extract complex temporal features. After that, it proposes the use of the adaptive exponential leaky integrate-and-fire (AdExLIF) spiking neuron. Finally, it described the capability of AdExLIF in sequentially processing images.

5.2.1 Challenge of Extracting the Global Context Utilizing the Temporal Dynamics of the Spiking Neuron

Extracting the global context using the temporal dynamics of the spiking neuron is challenging. This challenge has been addressed using various approaches in the SNN literature [64, 125, 128]. However, these approaches violate our design constraints of having small intermediate results in the intermediate layers. These approaches rely on the idea of extracting the global context using spatially wide network architectures. For example, in the self-attention mechanism approach [128], the local features are

extracted first (and maintained in memory), then these features are concatenated in QK^T matrices to spatially extract the global context. Even though the extracted local features may have a smaller size than the size of image patches, this does not improve the memory overhead of the SNN, especially with a large sequence of patches. In our approach, we are constrained by our design to extract the global context only using the temporal dynamics of the spiking neuron. Next, we analyze the temporal dynamics of the spiking neurons to understand their limitation in global feature extraction.

Recall: Q4. What is the relationship between dynamicity and memory efficiency?

The size of the intermediate layers increases the number of spiking neurons which increases the memory overhead. In image application, applying the image sequentially reduces the size of the intermediate layers, which results in a lower number of spiking neurons. This pushes us to utilize the dynamical properties of the spiking neurons (increasing the utilization of every spiking neuron in the SNN). In other words, utilizing the dynamicity of the spiking neuron reduces the memory overhead of SNNs.

5.2.2 Temporal Dynamics of LIF Spiking Neuron

The (LIF) neuron is a widely used neuron model in the SNNs. This neuron model is simple in terms of computations and memory requirements, making it desirable for simulation of SNNs. On the other hand, this computational simplicity comes at the expense of sacrificing the rich temporal dynamics of the spiking neuron, which makes them unsuitable for extracting global context temporally. To illustrate this, we analyze the spiking neuron as a digital system that has linear and non-linear temporal dynamics.

The linear temporal dynamics (sub-threshold dynamics) of the LIF neuron is modeled by the following ordinary differential equation:

$$\tau_u \frac{du}{dt} = -g_L u + I \quad (5.1)$$

where u is the membrane potential variable, g_L is the leakage constant, τ_u is the time constant and I is the (weighted) input current. These linear dynamics are digitized for deployment in digital hardware. For simplicity, we use Euler method to digitize as following:

$$u[t + 1] = u[t] + \delta \frac{du}{dt} \quad (5.2)$$

Substituting Eq. 5.1 in Eq. 5.2, and for simplicity assuming τ_u and δ equal to 1, we get:

$$u[t + 1] = (1 - g_L)u[t] + I[t] \quad (5.3)$$

This difference equation represents u before reaching a threshold value u_{th} . When u reaches u_{th} , u value resets non-linearly. For example, the reset can be setting u to zero. The linear frequency response of the spiking neuron can be extracted by representing the membrane potential in the z -domain as follows:

$$H(z) = \frac{z^{-1}}{1 - (1 - g_L)z^{-1}} \quad (5.4)$$

The transfer function shows that the linear dynamics of the LIF neuron has one pole at $1 - g_L$, which can be viewed as a low pass filter of signals (discrete-time multi-amplitude signals). Fig. 5.1 shows the frequency response of the linear dynamics of the LIF neuron. As can be seen from the figure, changing the g_L does not change the cutoff frequency of the neuron. One can utilize the nonlinear dynamics in the LIF to change the cutoff frequency of the neuron; e.g., using a higher u_{th} allows for moving the cutoff to lower frequencies. However, this is not applicable when the input is a composite of different frequencies, as a different u_{th} is required for a different number of composites. This shows that the LIF neuron is very limited in extracting temporal features. Therefore, an SNN relying on LIF neuron model requires spatial extraction of features to compensate for the limited temporal dynamics in the LIF neuron model.

5.2.3 Higher Complexity Spiking Neuron for Global Context Extraction

The LIF spiking neuron model is a simplified model of the biological neuron model. Biologically plausible models (such as the Hodgkin–Huxley model) are more complex in terms of the nonlinearity of the temporal dynamics and the number of state variables. In our work, our aim is to use a neuron model that has a complex temporal dynamics while having simple computational and memory requirements. The Adaptive Exponential Leaky Integrate-and-Fire (AdExLIF) neuron [11] is a spiking neuron model that is described by two state variables as follows:

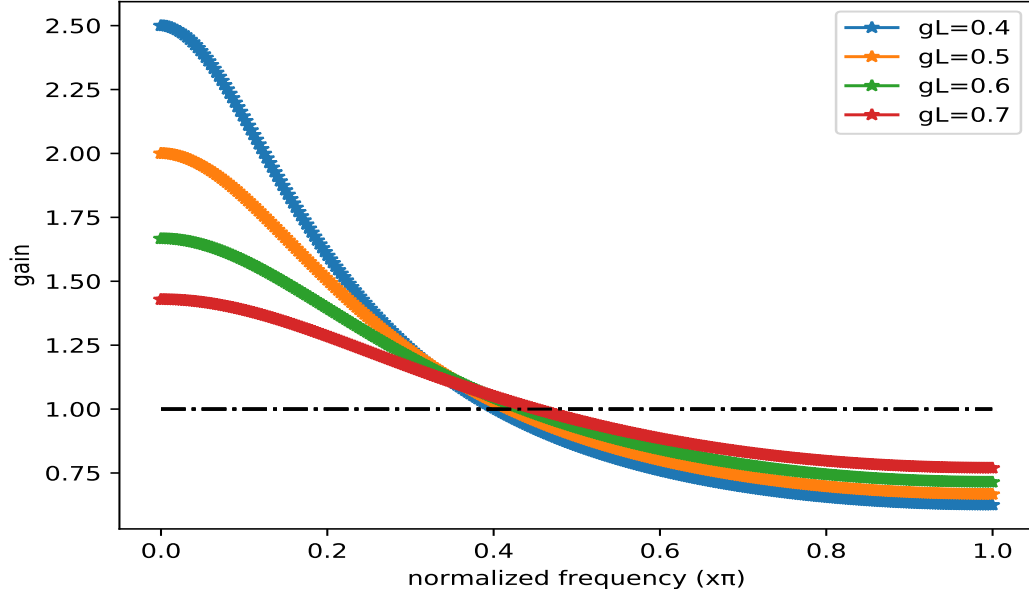


Figure 5.1: The theoretical linear frequency response of the LIF spiking neuron; the magnitude of the linear frequency response of the neuron.

$$\tau_u \frac{du}{dt} = -g_L u + g_L \Delta_T \exp\left(\frac{u - V_T}{\Delta_T}\right) - y + I \quad (5.5)$$

$$\tau_y \frac{dy}{dt} = a u - y \quad (5.6)$$

where y is the second state variable known as the adaptation variable, a is the adaptation constant, and $g_L \Delta_T \exp\left(\frac{u - V_T}{\Delta_T}\right)$ is the exponential growth of the membrane potential when it is close to the threshold value (V_T). In our experiments, we neglect the exponential term by setting V_T to a very high value. Note that we immediately reset u and y once u reaches the non-linear threshold u_{th} . As done for LIF neuron, the linear dynamics of the AdExLIF neuron is described in z-domain as follows:

$$H(z) = \frac{z^{-1}}{1 - (1 - g_L)z^{-1} + a z^{-2}} \quad (5.7)$$

The temporal linear dynamics of the AdExLIF neuron are richer than the LIF neuron. Fig. 5.2 shows the frequency response with varying a and fixing g_L . This shows that the AdExLIF works as a band pass filter that passes frequencies at different gains depending on the value of a . With proper choice of the nonlinear threshold u_{th} , this allows the AdExLIF to behave as a bandpass filter with different frequency

5.2. Proposed Solution: Efficient Processing Utilizing the Temporal Dynamics of the Spiking Neuron

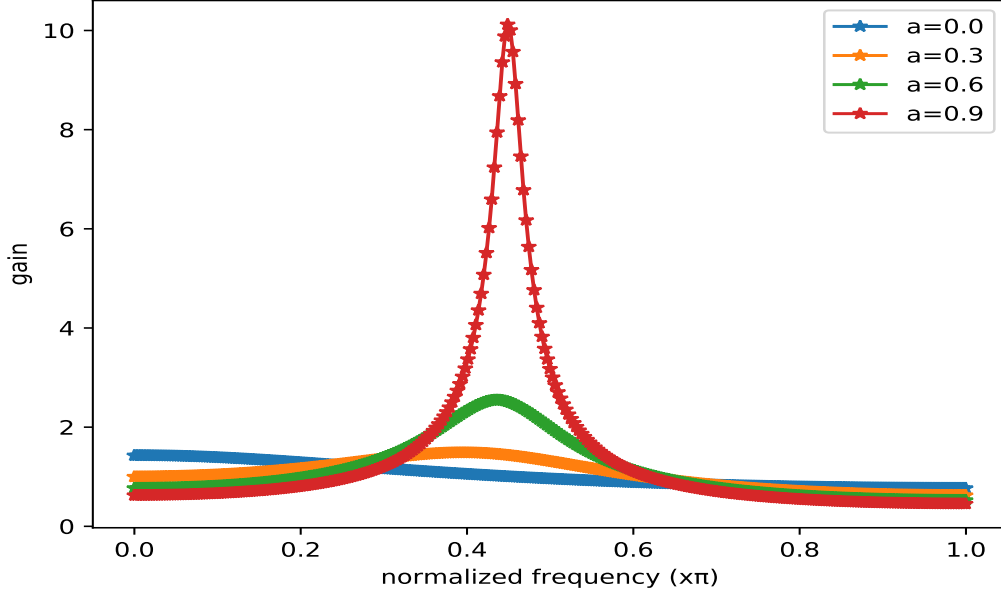


Figure 5.2: The theoretical linear frequency response of the AdExLIF spiking neuron. g_L was fixed to 0.8 and a is tuned from 0 to 0.9. This shows that AdExLIF can be tune to band widths of temporal features.

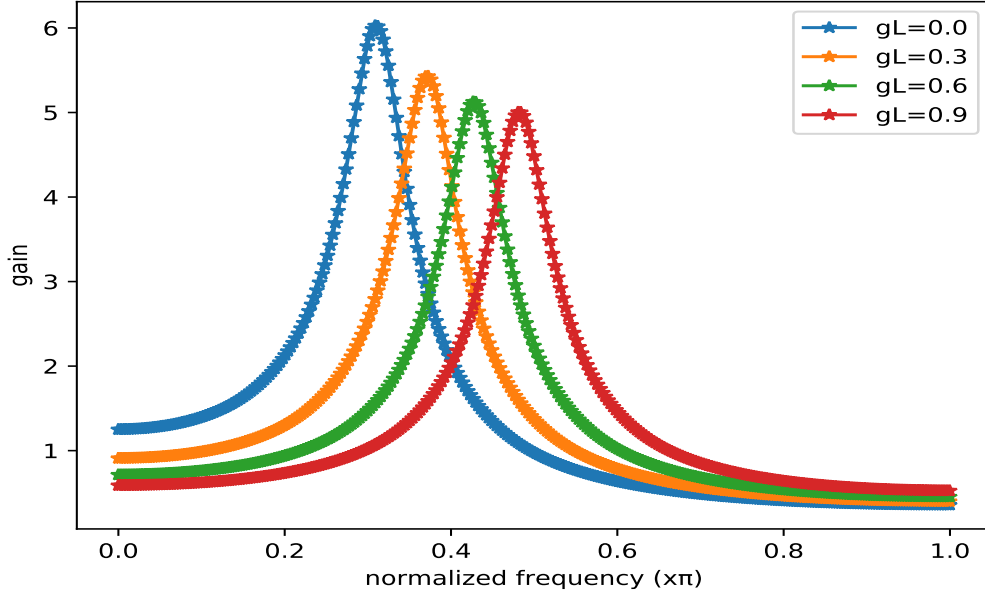


Figure 5.3: The theoretical linear frequency response of the AdExLIF spiking neuron. a was fixed to 0.8 and g_L is tuned from 0 to 0.9. This shows that AdExLIF can be tune to detect different ranges of temporal features.

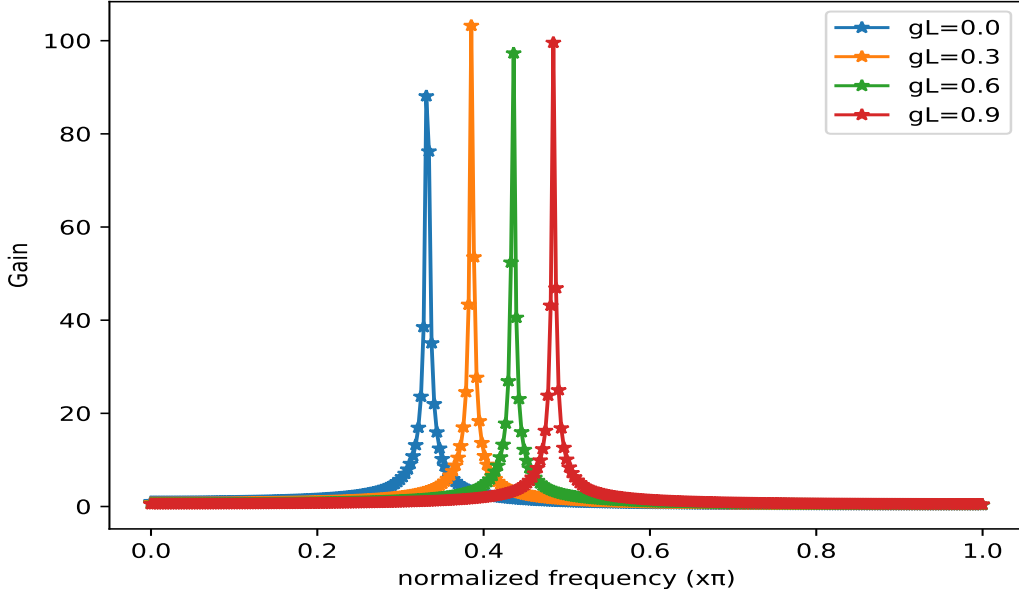


Figure 5.4: The theoretical linear frequency response of the AdExLIF spiking neuron. a was fixed to 0.99 and g_L is tuned from 0 to 0.9. This shows that AdExLIF can be tune to detect narrow range of temporal features.

bandwidths. Fig. 5.3 shows that the frequency band of specific a can be shifted to fit a desired band. Finally, 5.4 shows that with a values close to 1 (e.g., 0.99), the AdExLIF behaves as a notch filter, where very specific features can be detected by a single neuron. This diversity of linear temporal dynamics of the AdExLIF shows that a more biologically-plausible neuron is capable of extracting complex temporal features. In our work, we use this family of spiking neuron model to process images sequentially in patched way to improve the processing efficiency of the SNNs.

5.2.4 Sequential Processing of Images using AdExLIF

The main idea of our approach is to reduce the size of the intermediate results within a convolutional SNN by processing the image sequentially. In the most extreme case, one neuron can be used to process the whole image, where the image is applied pixel by pixel to the spiking neuron. Although few computational resources are used (one neuron), a relatively complex visual features can be extracted. To understand the capability of the AdExLIF neuron to extract visual features sequentially, we analyze its impulse response. Fig. 5.5 shows the main category of impulse responses of the AdExLIF neuron. The impulse response is simply produced by applying an impulse as

5.2. Proposed Solution: Efficient Processing Utilizing the Temporal Dynamics of the Spiking Neuron

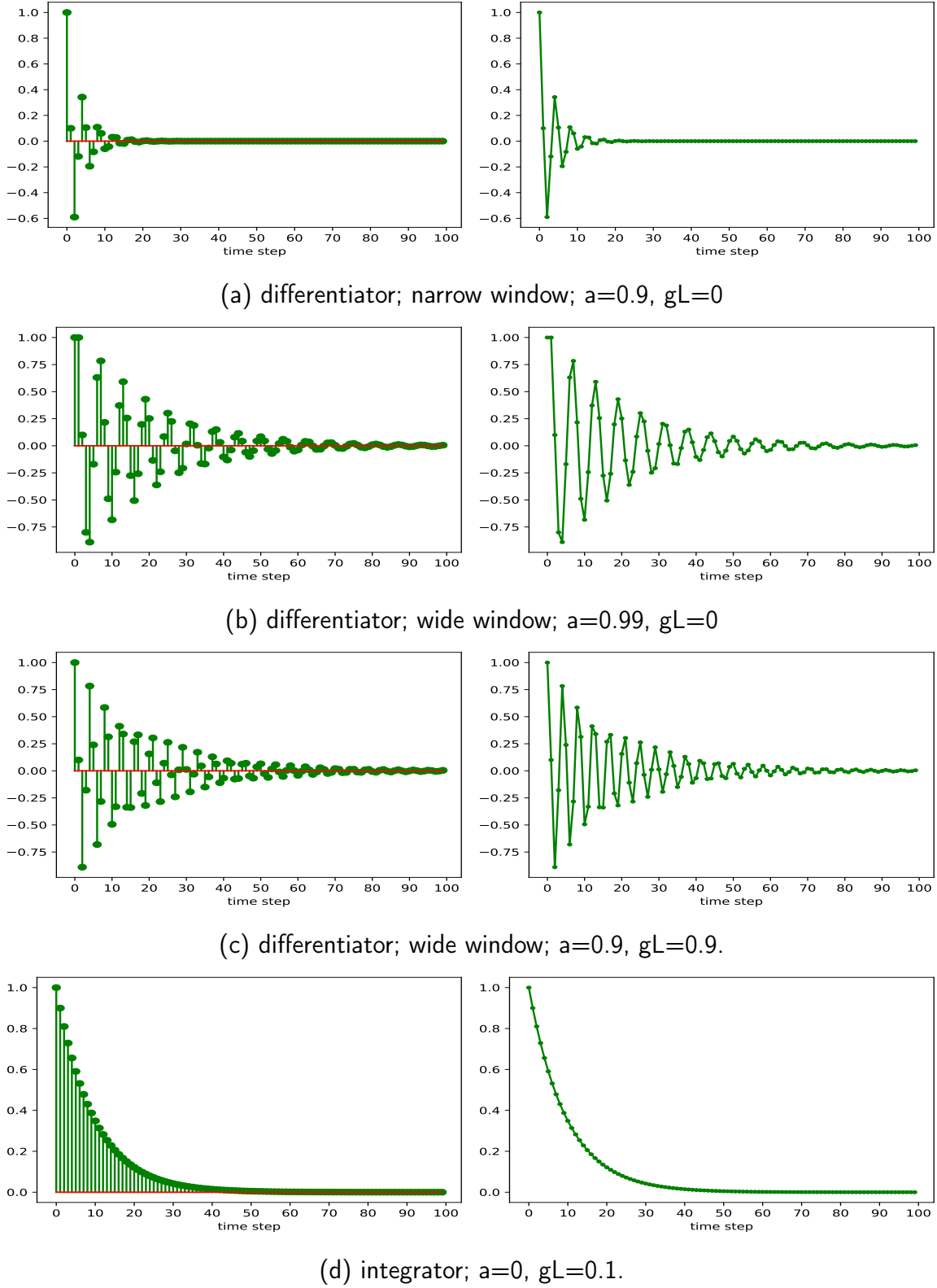


Figure 5.5: Different impulse responses (membrane potential response to an impulse) of the AdExLIF spiking neuron.

an input and measuring the membrane potential as an output. The linear response of the AdExLIF neuron is equivalent to convolving an input sequence with the impulse response. It is important to note that we do not convolve the sequential image with the impulse responses in Fig. 5.5, but this helps us understanding the theoretical potential of the spiking neuron in processing a sequenced image.

Fig. 5.5a, 5.5b and 5.5c show that the AdExLIF behaves as a differentiator. In Fig. 5.5a, the AdExLIF behaves as a narrow window differentiator. If an image is sequentially applied to this AdExLIF, pixel by pixel with the adjacent pixels in the same row being applied subsequently, the sequentially produced output is filtered image with amplified vertical edges. In Fig. 5.5b, this differentiator has a wide impulse response, allowing the AdExLIF to extract both vertical and horizontal edges. In this case, the vertical edges will have a higher amplitude than the horizontal ones. One can think of this neuron as an integrator of weighted image gradients, where the weights are exponentially decreasing. In other words, this AdExLIF passes the input image except some specific features. This has an opposite effect compared to the narrow window differentiator that passes only specific features while filtering out the rest of the features. Fig 5.5c shows also a wide differentiator but with a different frequency. This allows the AdExLIF to filter out different features from the image. Another type of responses is shown in in Fig. 5.5d. This AdExLIF behaves as an integrator, which allows filtering out high frequency features (an approximation of a gaussian filter).

It is worth mentioning that the types of the extracted features depend on the way the pixels of the input image are sequentialized. For example, applying the pixels sequentially row by row to a narrow window differentiator detects vertical edges, while applying them column by column detects horizontal edges. In addition, sub-sampling the input image affects the extracted features. Ideally, a sufficiently complex non-linear sub-sampler allows the AdexLIF to extract any visual feature sequentially. In this work, we restrict ourselves (for simplicity) to apply the image pixel by pixel in a horizontal way without any sub-sampler.

5.3 Experiments

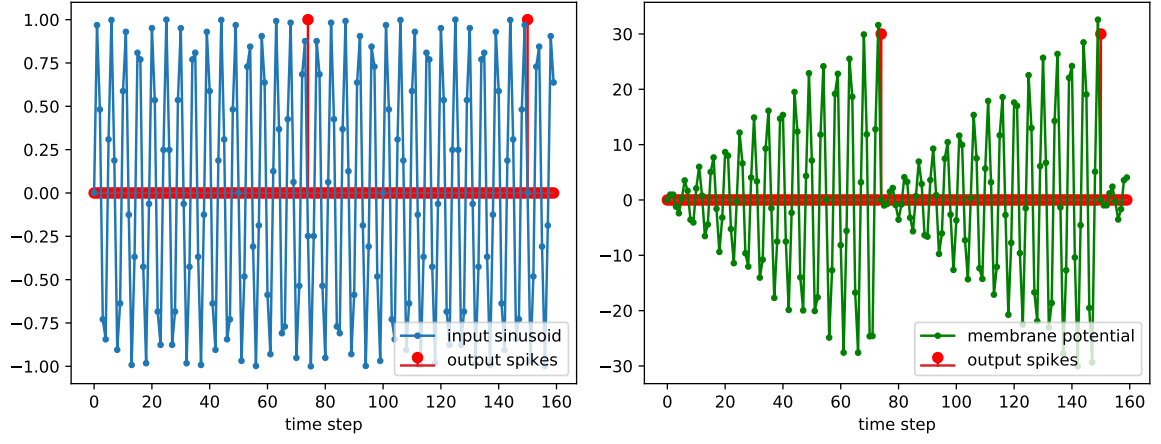
The presented results of this section are based on the AdExLIF spiking neuron. In all results the exponential term of the AdExLIF was neglected by setting V_T to 100. The LIF neuron is realized from the AdExLIF model by setting the adaptation constant (a) to zero. In all experiments, the input signal (whether it is a sinusoid or an image) is applied to the neuron sequentially; every sample point (e.g., pixel) is applied at every

time step to the neuron. Next, we present the results in detail.

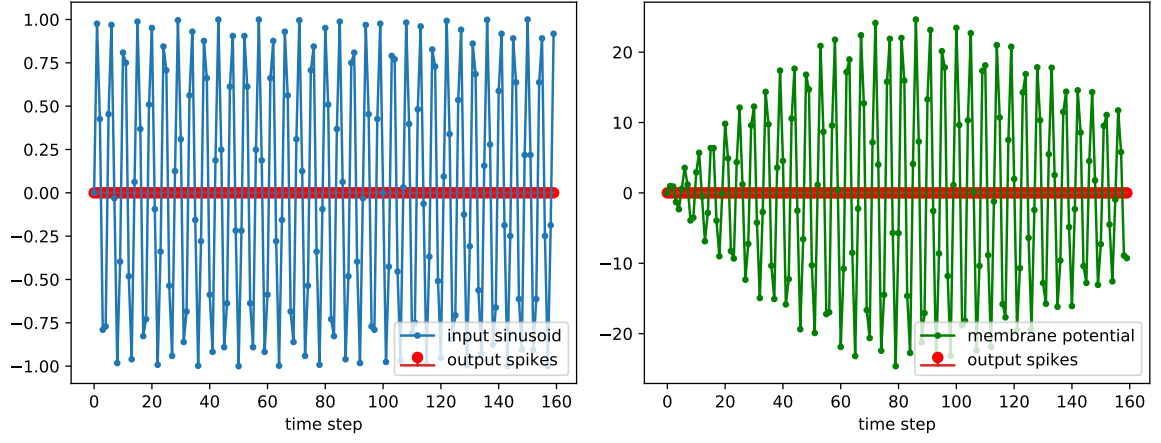
5.3.1 Transient Response

Fig. 5.5 shows the transient response of an AdExLIF neuron (parameterized with $g_L=0.5$, $a=0.99$ and $u_{th}=30$) at different sinusoidal input frequencies. This neuron was designed to notch a narrow range of frequencies (passes frequencies tightly around 210Hz). The figure shows the output spike, highlighted as a red impulse, with the input sinusoid (the constant oscillation which is highlighted in blue) and the membrane potential state variable (the varying-amplitude oscillation which is highlighted in green). Fig. 5.6a shows the neuron when it detects the input frequency. The neuron fires a spike after every number of sinusoidal cycles. This means one spike carries two information: the input frequency (that the neuron is designed to notch) and the number of cycles (that the neuron is also designed to detect). The number of cycles per spike can be used to represent the *frequency durability* (the amount of time a frequency component persists in a signal). The number of cycles per spike can be changed by changing the spiking threshold (u_{th}) of the neuron. This can be understood from the membrane potential, where the oscillation in the membrane potential increases until it reaches u_{th} and then resets. The increasing oscillation in membrane potential is caused by the adaptation, where the neuron tries to adapt to the constant sinusoid by increasing the adaptation state variable, which causes this increasing oscillation. Fig. 5.6b and 5.6c show the transient response of the neuron when the input sinusoid is out of the notched frequencies range. In both figures, the neuron does not output any spike. The membrane potential in both cases oscillates in increasing way then starts to damp. In fact, this is also caused by the adaptation variable, while in this case the maximum amplitude of the membrane potential does not reach u_{th} . This makes the membrane potential oscillate in both increasing and decreasing ways.

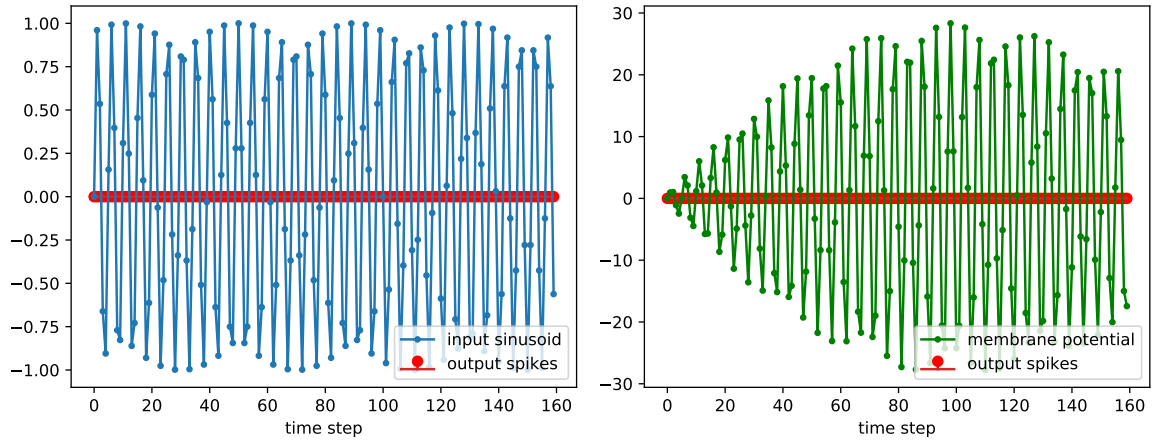
Fig. 5.7 shows (in comparison) the transient response of the LIF neuron. These parameters are the same as the previous parameters for AdExLIF except that the adaptation constant is set to zero ($a=0$). The LIF neuron is a one pole non-linear filter that could behave as a low pass filter. In this example, we chose a passed frequency component to compare the LIF transient response with the AdExLIF. The first main difference can be seen in what a spike represents in the two neurons. In LIF, if the input sinusoid is within the neuron's pass range, the neuron outputs a burst of spikes at every cycle. This shows that the AdExLIF neuron has a sparser representation of extracted information. In addition, a spike in AdExLIF neuron can be used to represent signal



(a) input frequency = 210Hz



(b) input frequency = 215Hz



(c) input frequency = 205Hz

Figure 5.6: Transient response of an AdExLIF neuron parameterized with $g_L=0.5$, $a=0.99$ and $u_{th}=30$. The neuron resets the membrane potential and adaptation variables to zero when outputting a spike. 73

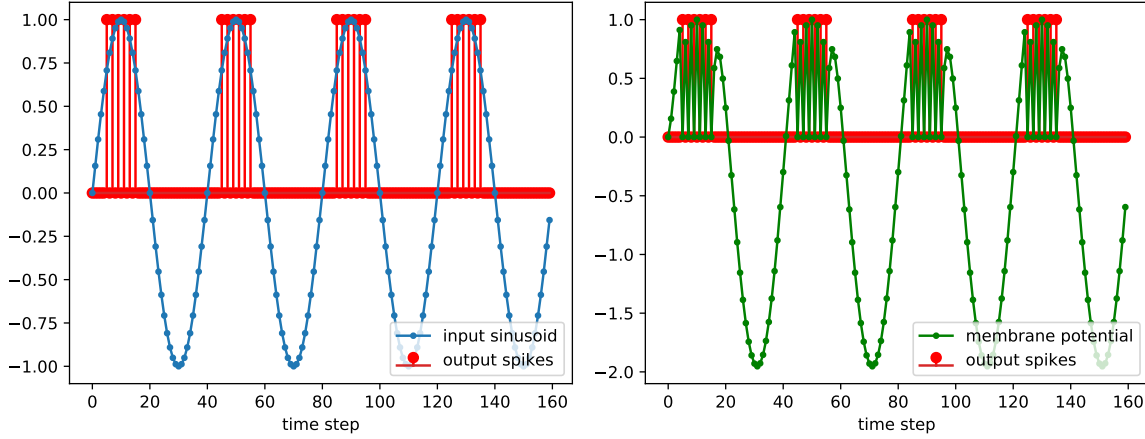


Figure 5.7: Transient response of the LIF neuron parameterized with $g_L=0.5$ and $u_{th}=0.8$. The neuron resets the membrane potential and adaptation variables to zero when spike.

durability, while this is not possible with one LIF neuron. On the other hand, the LIF neuron can be more robust to errors such that a single sinusoidal peak is represented by a burst of spikes.

Fig. 5.8 shows the transient response of the AdExLIF neuron (parameterized at $a=0.8$, $g_L=0.8$, $u_{th}=2$) for a long period of time. When the neuron operates in the pass mode (in the range between $\tilde{175}\text{Hz}$ to $\tilde{275}\text{Hz}$), the neuron's spiking output is periodic. This means that the AdExLIF transforms a periodic sinusoidal signal to a periodic spiking signal. The frequency of the output signal is linearly proportional to the frequency of the input signal. In fact, this shows that the AdExLIF neuron can operate as a compressor (a modulator) of single-frequency signals. In addition, at the cutoff frequencies of the AdExLIF neuron, the neuron behaves unexpectedly for a short range of frequencies. This is due to the dynamical complexity of the AdExLIF neuron. From engineering perspective, this degrades the reliability of the produced outputs in AdExLIF. In the next section, we deal with this problem by increasing the adaptation constant and using multiple neurons.

5.3.2 Frequency Response

Fig. 5.9 shows the frequency response of ten AdExLIF neurons that vary in the leakage constant (g_L). To test the temporal frequency extraction capability of the AdExLIF, we decided to use a relatively narrow band AdExLIF. Therefore, we used a high adaptation variable ($a = 0.99$) and a high firing threshold ($u_{th}=30$). The sampling frequency of

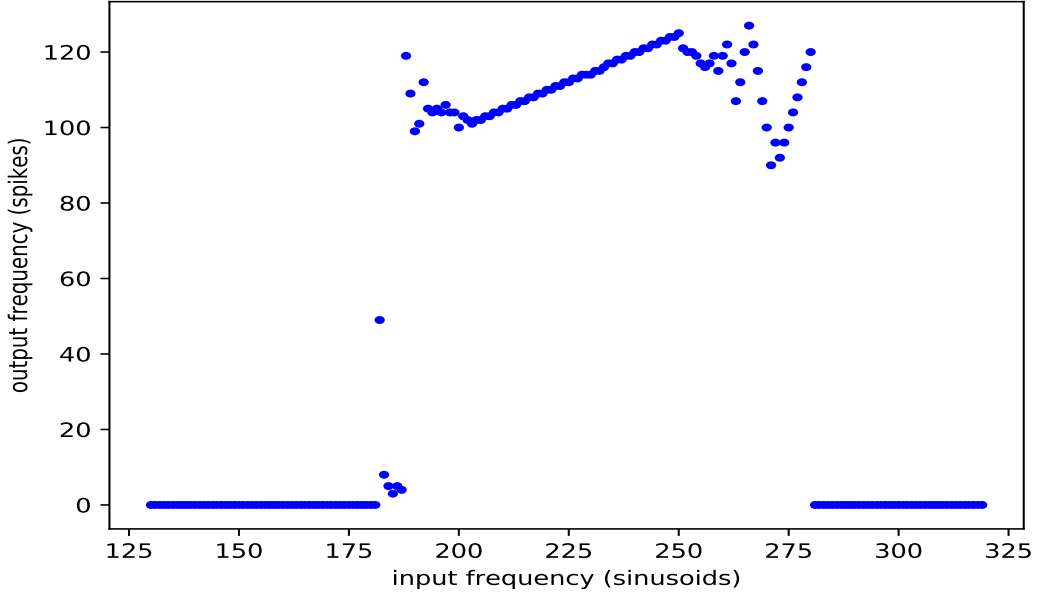


Figure 5.8: Input frequency to output frequency mapping of AdExLIF (parameters: $a=0.8$, $g_L=0.8$, $u_{th}=2$). This shows the transient response of an input sinusoid that persists for long period of time. The neuron spiking output is also periodic, where an a periodic sinusoidal signal is transformed to a periodic spiking signal.

the input sinusoid is 1000Hz. As can be seen from the figure, the 10 AdExLIF neurons detect 10 different narrow frequency bands. In this example, these frequency bands were manually chosen not to overlap with each other; the width of this band can be changed by tuning a and u_{th} . These 10 neurons can be seen as a frequency decomposer. The detection range of this frequency decomposer is limited between 150Hz and 250Hz, with 10 different frequency bands.

We extended the detection capability of this frequency decomposer by shifting the frequency of the input signal by a cosine mixer as follows:

$$\begin{aligned} \cos(2\pi f_{in}) \cos(2\pi f_{shift}) = \frac{1}{2} [& \cos(2\pi(f_{in} - f_{shift})) \\ & + \cos(2\pi(f_{in} + f_{shift}))] \end{aligned} \quad (5.8)$$

where f_{in} is the input frequency and f_{shift} is the shift value in the input signal. Note that this is a double side shift, where the input is shifted in positive and negative directions. However, this can be neglected as we can choose the operating range of the

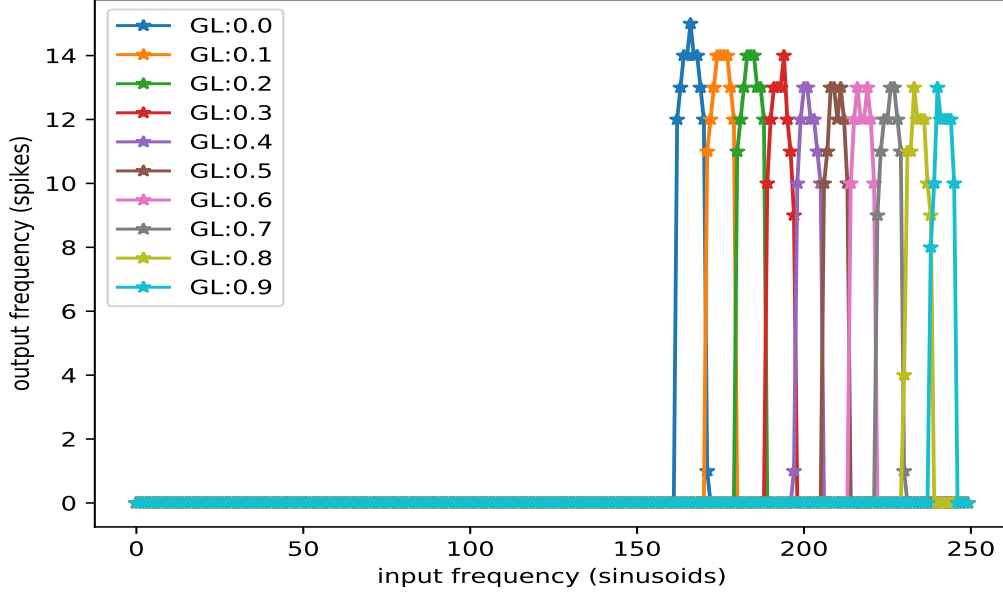


Figure 5.9: experimental frequency response of a frequency decomposer example; AdExLIF: $a=0.99$, $u_{th}=30$, $V_T=100$

decomposer to be $[0, 0.25F_{sample}]$.

Fig. 5.10 shows the frequency response of a frequency decomposer made of 25 AdExLIF neurons. The dection range of this decomposer is between $[0, 250\text{Hz}]$ (with a sampling frequency of 1000Hz). The first 10 neurons are the same as the base frequency decomposer (shown in Fig. 5.9) but with shifting the input signal to all these neurons by $0.16F_{sample}$ (160Hz). The next five neurons are the same as the first five neurons in the base frequency decomposer but shifted by $0.0897F_{sample}$ (89.7Hz).

These results show the richness of the temporal dynamics of the AdExLIF neuron. The AdExLIF neuron computationally differs from the LIF neuron by only one state variable (the adaptation variable), while it functionally has a much richer temporal dynamics.

5.3.3 Sequential Convolver of Images

We applied images to the AdExLIF sequentially as a 1D signal, such that a single AdExLIF neuron is used to filter a whole image. Our experiments show that by only varying the leakage constant (g_L) and the spiking threshold (u_{th}), we can extract vertical edges and horizontal edges. Fig. 5.11 shows the linear impulse response of the 10

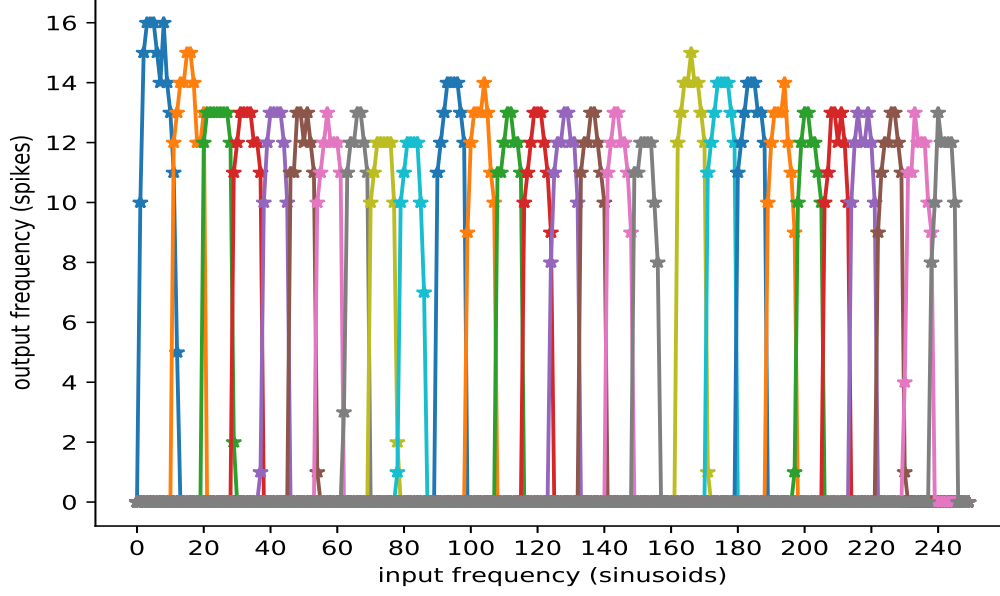


Figure 5.10: experimental frequency response of a frequency decomposer example; 3 decomposers; cosine mixer (input frequency shifter); AdExLIF: $a=0.99$, $v_{th}=30$, $V_T=100$

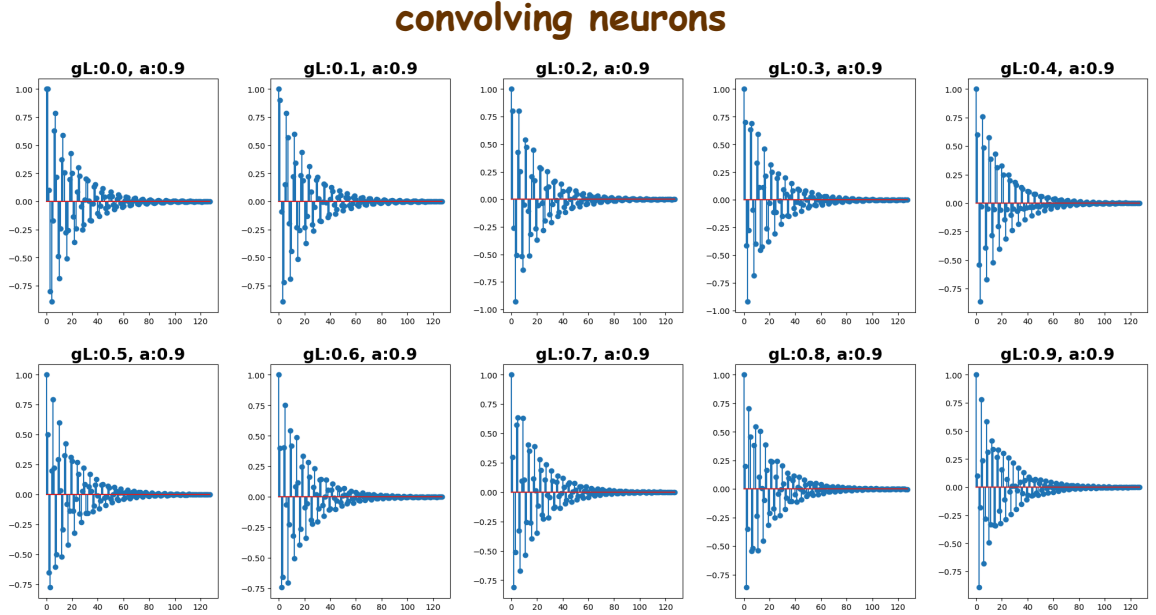
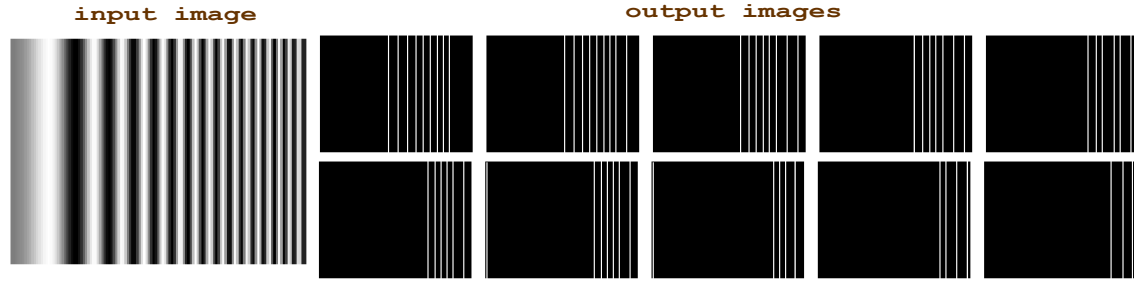
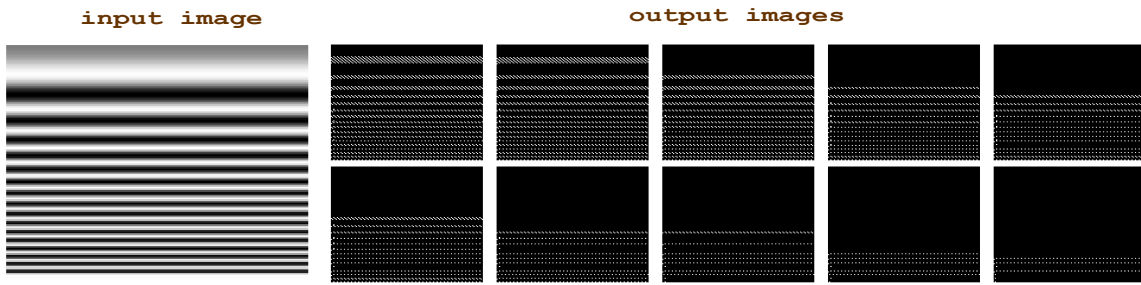


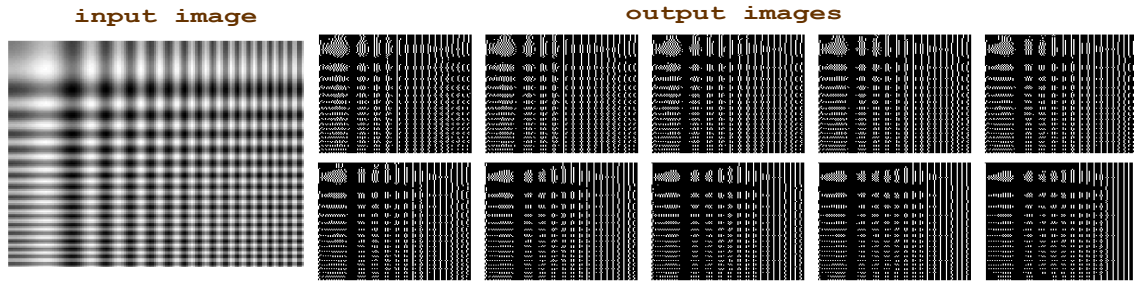
Figure 5.11: impulse responses of 10 different neurons, parameterized with fixed a ($=0.9$) and varying gL from 0 to 0.9



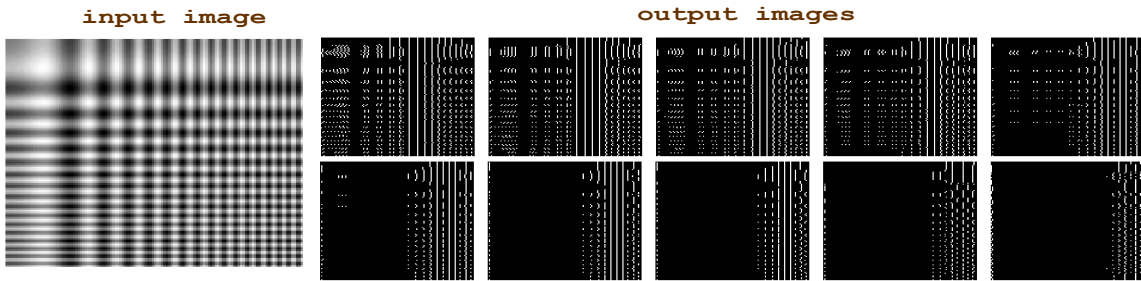
(a) $u_{th}=2$



(b) $u_{th}=1$



(c) $u_{th}=1$



(d) $u_{th}=2$

Figure 5.12: Image processed sequentially (pixel by pixels) by the neurons with impulse responses in previous Fig. 5.11 (location of the output image is filtering with the neuron with the corresponding location in Fig. 5.11).

AdExLIF neurons used in our experiments.

Fig. 5.12 shows different images filtered sequentially by the AdExLIF neuron. We used three different images to visualize the feature extraction capability of AdExLIF: sinusoidal frequency sweep in the vertical direction, sinusoidal frequency sweep in the horizontal direction, and sinusoidal frequency sweep in 2D. Each output image is processed by one AdExLIF neuron, with the parameters shown at the corresponding location in Fig. 5.11.

Fig. 5.12a shows the detection of vertical lines at different frequencies. This figure shows that neurons of different parameters is capable of detecting different features. In all output images, the low frequency components are blocked. The neurons with higher g_L detects edges of higher frequencies (thinner lines). Fig. 5.12a shows the detection of horizontal lines at different frequencies. In this case, we can see that the low frequency features are passes by some neurons. Even though the neurons here have the same parameters as in the image of vertical lines, the orientation of the lines in the image is different. This shows the importance of the way pixels are applied to the neuron (in our experiments, pixels are applied sequentially in the horizontal direction). This can be observed better in Fig. 5.12c and Fig. 5.12d. In these figures, frequency components in both directions are merged in the same input image. The only difference between Fig. 5.12c and Fig. 5.12d is the non-linear threshold u_{th} . When u_{th} threshold is higher, the neurons tend to pass higher frequencies. In addition, in both figures, the passed frequencies are amplified or attenuated depending on their orientation. The passed vertical frequencies are amplified, while the passed horizontal lines are attenuated. The attenuation of horizontal lines increases when increasing u_{th} . Note that by superposition (linear combining the output images from different neurons), one can extract more complex visual features.

Fig. 5.13 shows the impact of the non-linear dynamics of the AdExLIF neuron on the response of the filter. Fig. 5.13a, the neuron has a higher threshold that prevents the neuron from firing. You can see how the impact of IIR filtering (with a wide impulse response) distorts the visual information in the image (see the linear response in membrane potential). In Fig. 5.13b, the neuron has a lower u_{th} which allows the neuron to spike and reset the membrane potential. This non-linear resetting of membrane potential maintains the visual information and hence stabilizes the neuron response. This makes the spiking neuron attractive in comparison to the known linear IIR filters.

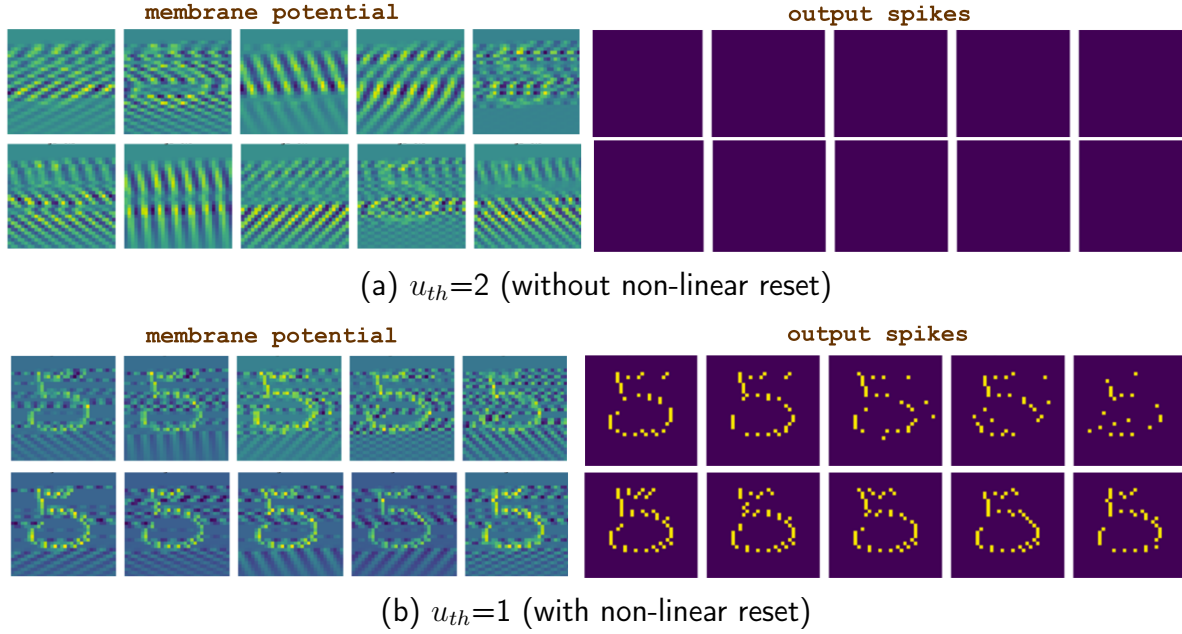


Figure 5.13: Transient response: AdExLIF neuron, $g_L=0.5$, $a=0.99$, $u_{th}=30$, reset to zero when spike

5.4 Conclusion

This chapter presented the capability of the spiking neuron to process signals in a temporal (sequential) fashion. The chapter showed the limitations of the LIF spiking neuron to extract complex temporal features. Then, the use of AdExLIF as a spiking neuron with more complex temporal dynamics were proposed. The results showed that AdExLIF is capable of extracting a wide range of information from temporal signals such as a narrow range of frequency, frequency duration, sinusoidal to spiking frequency conversion and sequential image processing. This allows the AdExLIF neuron to be utilized in a more flexible network architectures that allows us to design a memory efficient networks. The next chapter utilizes the dynamical complexity of this type of spiking neurons to design memory-efficient SNN architectures.

Chapter 6

Spatiotemporal Convolutional Spiking Neural Networks for Memory-Efficient Processing

This chapter aims to exploit the dynamical properties of the spiking neuron to structure SNNs to process static images in a memory-efficient way (answer to Q5). We propose a spatiotemporal convolutional approach that processes static images as a sequence of 2D patches. We show that processing images as a sequence of 2D patched images allows for better use of the hardware platform based on its memory resources. Our results show that the temporal dynamics of the spiking neuron is a unique feature in SNNs, which play an important role in their memory overhead and hence processing efficiency. The content in this chapter is included in the following paper: Muath Abu Lebdeh et al. “Memory-Efficient Spatiotemporal Convolutional Spiking Neural Networks for Vision Applications,” paper under submission to the IEEE Transactions in Emerging Topics in Computational Intelligence.

6.1 Introduction

The previous chapter shows the dynamical richness of complex spiking neurons in extracting spatial features. This dynamical property increases memory overhead at the neuron level. On the other hand, this property can be utilized to process images in a spatiotemporal way, allowing for memory efficient networks.

As discussed in earlier chapters, the dynamical property of the spiking neuron was viewed as a memory overhead, which requires additional techniques and approaches to

reduce it. In this chapter, we exploit the dynamicity of the spiking neuron to reduce memory overhead without introducing any spatial architectural ideas such as those described in the literature [56, 57, 48, 38, 38, 85, 73, 105]. Some works studied the dynamicity of the spiking neuron and applied it to different applications [77, 36, 27, 94, 42, 3, 66]. However, these solutions do not investigate the relationship between the dynamicity of the spiking neuron and the memory overhead.

In this chapter, we propose an approach to reduce the memory overhead of the SNN in vision applications by utilizing the dynamicity of the spiking neuron. The proposed approach uses a spatiotemporal convolutional block to process images in a sequential manner. This reduces the memory overhead of SNNs and improves their processing efficiency depending on the hardware memory resources. The contributions of this chapter are as summarized following:

- We propose a spatiotemporal convolutional network that processes an image sequentially utilizing the intrinsic properties of the spiking neuron.
- We present empirical findings on the trade-offs that sequential processing in SNN provides under different hardware memory resources.

6.2 Proposed Solution: Spatiotemporal Convolutional SNN

6.2.1 Proposed Network Architecture

This section presents the proposed network architecture. It starts by describing the proposed network architecture and justifying the design choices. Then, it presents the restrictions imposed on the AdExLIF neuron to stabilize the network training using the BPTT.

As the memory overhead is the main bottleneck, especially in SNNs, we aim to designing a network architecture that reduces the memory overhead. In this context, the memory overhead is improved in terms of utilization of a hierarchical memory system, the maximum runtime memory and the offline storage requirement. To ensure that an SNN has a reduced memory overhead, we impose the following restrictions on the network design:

- **Reduce the size of feature maps:** In convolutional SNNs (CONV-SNNs), every intermediate layer has several feature maps as inputs and several feature

maps as outputs. Every output feature map requires processing of all input feature maps. In other words, every input feature map is re-used for the next output feature map only when all other input feature maps are processed for the current output feature map. The size of the memory required to hold all input feature maps is the reuse distance which is inversely proportional to the utilization of a hierarchical memory system. Therefore, reducing the size of the feature maps in the intermediate layers of a CONV-SNN improves the utilization of hierarchical memory, potentially improving the processing efficiency of the SNN. In addition, reducing the size of the feature maps results in a reduced number of spiking neurons. This has an impact on runtime memory and on the utilization of the memory hierarchy.

- **Use the inherent properties of the SNN:** At the neuronal level, the SNN is computationally complex enough compared to standard ANNs. Instead of borrowing architectural concepts from ANNs, we aim to utilize the dynamicity of the spiking neurons to process data temporally. In other words, we do not want to introduce any (spatial) architectural biases to the SNN, except using the convolutional kernel as a famous architectural bias for vision tasks. This allows us to use the functional properties that a spiking neuron offers rather than keeping them idle. Note that here we also want to maintain the SNN capability of integrating (spatial) architectural concepts from ANNs. Therefore, we choose the CONV-SNN as a use case of our proposed approach.

Below we describe the design choices made with restrictions mentioned previously:

- An input image is processed in spatial and temporal dimensions. The 2D image is divided into smaller 2D patches: E.g., in Fig. 6.1, 28x28 image is divided into 7x7 patches. In this example, the total number of patches is 16.
- As the CONV-SNN is capable of processing temporal data, one patch is applied at every step as shown in Fig. 6.2. In our example, the CONV-SNN processes the image in 16 steps (a 7x7 patch is applied in every step).
- Every patch is processed spatially by the spatial convolutional kernel, while the spiking neuron temporally processes the sequence of patches as shown in Fig. 6.3. This means that the spatial convolution kernel extracts the local features in the image, while the spiking neuron extracts the global contexts from the spatially extracted local features.

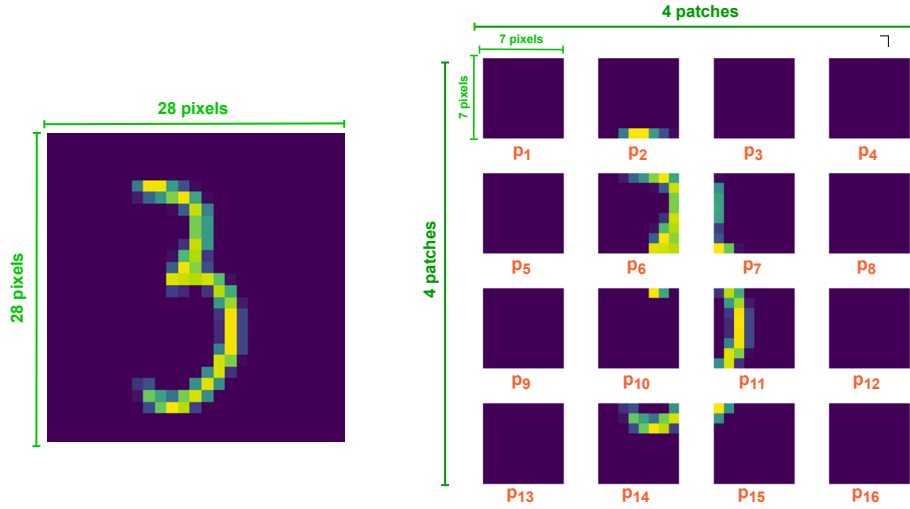


Figure 6.1: An example of patching an MNIST image. This 28x28 image is patched into 4x4 (16) patches, each of 7x7 (49) pixels. The local feature in every patch is extracted spatially, while the global features between different patches are extracted temporally.

- Computationally, allowing the spiking neuron to extract the global context in the image reduces the memory overhead. At every step, the intermediate layers have smaller results in terms of the memory required. This results in a reduced runtime memory, which reduces the cost of the host hardware. In addition, the smaller size of each feature map means that the number of neurons used in every layer is reduced, which reduces the number of the trainable parameters in the SNN model. This also reduces the runtime memory required to store the spiking neurons' state variables. Finally, the reduced size of the intermediate results allow them to fit better into local memories, and hence utilize better a hierarchical memory system.
- Therefore, the proposed solution is expected to improve the overall processing efficiency the SNN as well as reducing their cost when deployed on hardware.

In this solution, we are constrained by our design to extract the global context only using the temporal dynamics of the spiking neuron.

Recall: Q5. How the dynamicity of the SNN can be utilized to reduce the memory overhead?

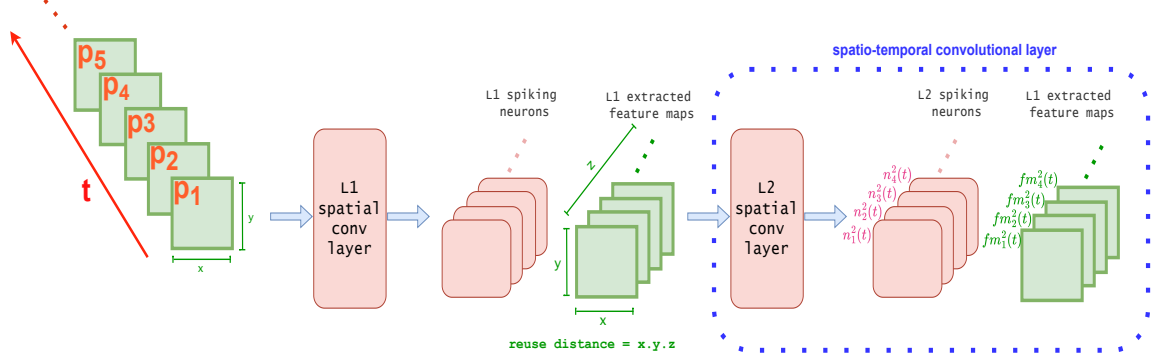


Figure 6.2: The proposed spatiotemporal convolutional SNN (CONV-SNN) architecture processes an input image as a sequence of 2D patches (tokens). This reduces the memory required for the intermediate results. This architecture extract the global visual context temporally, utilizing the intrinsic temporal dynamics of the spiking neuron.

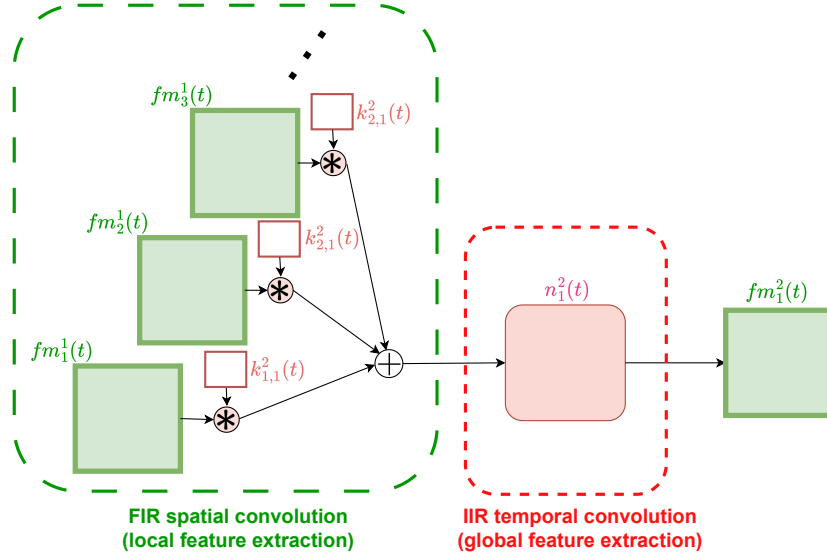


Figure 6.3: The spatiotemporal spiking convolutional layer consists of two types of convolutions. The FIR spatial convolution is used to extract the local context from every 2D patch. The IIR temporal convolution is used to extract the global context from the spatially processed sequence of 2D patches.

The proposed CONV-SNN process an image as a sequence of patches. The dynamicity of the spiking neurons are utilized to extract the global context in a temporal manner. This increases the utilization of the spiking neurons, reducing their number, and hence reducing the overheadll memory overhead.

6.2.2 Network of AdExLIF

As disscussed in the previous chapter, a simple neuron model such as LIF neuron is not sufficient to extract complex global contexts without using a spatial architectures. Therefore, we use the AdExLIF neuron model as model of relatively complex temporal dynamics. The numerical approximation of the AdExLIF has 4 parameters (excluding the parameters of the exponential term): g_L , a , u_{th} , δ . The AdExLIF is a dynamical non-linear IIR system that could have a set of parameters that result in instable response. Therefore, some boundaries are needed to guarantee a stable response in the network. In addition, this task becomes more challenging when integrating AdExLIF neurons in a network.

To address this problem, we adopt the boundary conditions proposed in [36]. First, this approach uses Resonate-and-Fire (RF) [41] neuron model, which can be seen as a conditioned AdExLIF neuron. The state variables of the neuron are described as follows:

$$\frac{du}{dt} = -bu - \omega_0 y \quad (6.1)$$

$$\frac{dy}{dt} = \omega_0 u - by \quad (6.2)$$

where u and y are the state variables (the membrane potential and the adaptation variable), ω_0 is the resonance angular frequency of the neuron, b is the damping factor. The RF neruon can be considered as the AdExLIF but with restriction on the neuron's parameters. This restriction allows the neuron to extract specific features in predictable way. Such that w directly represents the angular resonating frequency (the neuron natural frequency or the target frequency that the neuron extracts).

In [36], the balanced RF (BRF) neuron was proposed with further conditios on the RF neuron to guarantee a stabilized response in a network. First, the threshold of the RF neuron is modified by a refractory period term, such that the threshold increases when the neuron fires. This allows the RF neuron to spike less redundantly. Second, the

Table 6.1: Performance comparison of different models.

Dataset	Model	Architecture	No. params.	No. steps	SOPs* ($\times 10^6$)	SOPs/step	Test acc.
Sequential MNIST	RC-SNN[36]	(1x1),256r,10f	68,874	784	3.55	4,533	97.86%
		(1x1),256r,256r,10f	200,458	784	10.43	13,301	98.00%
		(1x1),256r,256r,256r,10f	332,042	784	18.86	24,060	97.43%
	FF-SNN	(1x1),256f,10f	3,338	784	0.02	21	90.17%
		(1x1),256f,256f,10f	69,386	784	0.80	1,024	96.33%
		(1x1),256f,256f,256f,10f	135,434	784	3.58	4,570	97.51%
		(2x2),32c,p,10f	1,866	196	0.02	102	92.43%
		(2x2),32c,32c,p,10f	11,370	196	0.70	3,571	96.37%
		(2x2),32c,32c,32c,p,10f	20,874	196	1.44	7,346	96.32%
	CONV-SNN	(4x4),32c,p,10f	26,34	49	0.01	204	91.7700%
		(4x4),32c,32c,p,10f	12,906	49	0.81	16,530	95.05%
		(4x4),32c,32c,32c,p,10f	23,178	49	1.55	31,632	95.63%
		(7x7),32c,p,10f	4,746	16	0.01	625	89.46%
		(7x7),32c,32c,p,10f	17,130	16	0.84	52,500	91.59%
		(7x7),32c,32c,32c,p,10f	29,514	16	2.15	134,375	93.82%
		(14x14),32c,p,10f	14,154	4	<0.01	<2,500	85.07%
		(14x14),32c,32c,p,10f	35,946	4	0.91	227,500	90.55%
		(14x14),32c,32c,32c,p,10f	57,738	4	1.96	490,000	92.53%

Under the Architecture column: f indicates a fully connected layer, r indicates recurrent fully connected layer, c indicates a spatial convolutional layer, p indicates an adaptive pooling of output size (2,2) and (mxm) indicates the dimension of the applied data in every step.

BRF uses a soft reset, where the damping factor changes as a function of the refractory period. Finally, the parameters of the BRF neuron (w_0 and b) are bounded to guarantee a converging membrane potential. The next section shows results on using this neuron model in sequential CONV-SNN.

6.3 Experiments

This section presents the evaluation of the proposed architecture. First, it compares the proposed architecture with other architectures from literature in terms of accuracy, number of parameters, the number of spiking operations and the number of steps. Then, it shows the gains of the proposed approach in terms of the maximum runtime memory. Finally, this section shows latency and throughput result when running the proposed network architecture on CPU and GPU.

6.3.1 Accuracy and Theoretical Efficiency

Table 6.1 compares different sequential architectures on (sequential) MNIST dataset. Three types of architectures are used: recurrent SNN (RC-SNN) [36], feedforward SNN (FF-SNN) and convolutional SNN (CONV-SNN). In RC-SNN and FF-SNN, the input image is sequentially applied pixel by pixel to the network. In CONV-SNN, the input image is sequentially applied patch by patch. The smallest patch has the size 2x2. The

size of the applied patch in every step in the table. For every network type, different depths are used, from one to three layers.

Accuracy. The recurrent architecture performs the best in terms of accuracy. The FF-SNN and CONV-SNN of 2x2 patches have close test accuracy. The accuracy of the CON-SNN decreases as the patch size increases. This was unexpected as the number of parameters increases with increasing the patch size. However, with increasing patch size, more data are applied to the parameters of the spatial convolution kernels, whereas less data are applied to every spiking neuron. This shows a trade-off on what parameters receive more data to be trained on. The accuracy results show that the temporal neuron parameters of the CONV-SNN are more powerful than the synaptic spatial parameters.

Number of parameters. Under the same architectural depth, the tested CONV-SNNs have less parameters than both the RC-SNNs and the FF-SNNs. The CONV-SNNs are more efficient in terms of parameters by a factor of 2.34x to 15.9x. In the CONV-SNNs, the number of parameters increases as the number of neurons to process each feature map increases; i.e., one neuron is used per feature map element. For example, 2x2 neurons are used per feature map with 2x2 patch size, while 14x14 neurons are used per feature map with 14x14 patch size. As will be shown in a next subsection, the number of parameters does not necessary reflect the processing efficiency of the network architecture when running on the hardware. However, the number of parameters affects the training time. In addition, the number of parameters does not reflect the maximum runtime memory as will be shown in a next subsection, but they reflect the storage required outside execution.

Number of steps. The number of steps equals to the number of times the network is iterated. For longer sequences, the network is iterated more. For example, in RC-SNN, the number of steps is 784 as one pixel is applied at a time, while it is 4 in the CONV-SNN of 14x14 patch size as 196 pixels are applied at every step. In fact, as will be shown in a next section, the number of steps also do not reflect the latency when the network runs on the hardware. This depends on the number of operations in the network architecture in all steps, the network’s utilization of hierarchical memory, and the intermediate memory of the host hardware. On the other hand, the number of steps affects the training time due to the parallelism capability of every network architecture. For example, CONV-SNN of 14x14 input patches trains faster than the CONV-SNN of 2x2 input patches even though it has more parameters. The reason for that higher level of parallelism allowed in the CON-SNNs of larger patches is higher.

Total spiking operations (SOPs). This is a theoretical estimate of the *energy* efficiency of the SNN when running on neuromorphic hardware; i.e., assuming all neurons

of the SNN are fabricated on the hardware. As shown below, SOPs is calculated for every layer by multiplying the number of spikes in every layer by the number of updated neurons in the next layer:

$$SOP_s = \sum_{\forall l} S^l \cdot n^{l+1} \quad (6.3)$$

S^l is the number of spikes in layer l and n_{l+1} the number of updated neurons by a spike in layer $l + 1$. n_{l+1} is different for different network architectures; e.g., in FF-SNN, a spike updates all neurons in the next layer, in RC-SNN, a spike updates all neurons in the current and the next layer, in a CONV-SNN with an FIR convolution kernel of size 3x3, a spike updates 9 neurons in every feature map in the next layer.

For the same network depth, the CONV-SNNs have the lowest SOPs compared to FF-SNN and RC-SNN. In network architectures of two layer depth, the SOPs of the CONV-SNN outperforms the RC-SNN by two orders of magnitude. The CONV-SNN also outperforms the FF-SNN architectures by around 2x with architecture depth of 3 layers. For the CONV-SNN of different patch sizes, the SOPs increases on average with the increase in the patch size. Note that these SOPs values do not reflect the energy efficiency in systems with hierarchical memory system; i.e., hardware systems that do not have memory and compute resources equal the number of neurons and the intermediate results size.

SOPS/step. Similar to SOPs, this is a theoretical estimate of *power consumption* on a network architecture on ideal neuromorphic hardware. With the same network depth, RC-SNN has more SOPs than FF-SNN whereas both networks have the same neurons and the number of steps. This makes the SOPs/step in FF-SNN lower than the RC-SNN. Except for patch size of 2x2, the CONV-SNN has higher SOPs/step compared to RC-SNN and FC-SNN. Even though the CONV-SNN architecture has on average lower SOPs than the other architectures, the impact of lowering the number of steps exceeds the reduction in the SOPs. Similar effect appears when increasing the patch size of the CONV-SNN; the higher the patch size, the higher SOPs/step.

Summary of Table 6.1 results. Below are the key conclusions of the results described above:

- *Accuracy:* The CONV-SNN architecture has some accuracy drop of at least 1% compared to the RC-SNN architecture.
- *Number of parameters:* The CONV-SNN architectures require less parameters than other architectures, improving the training speed.

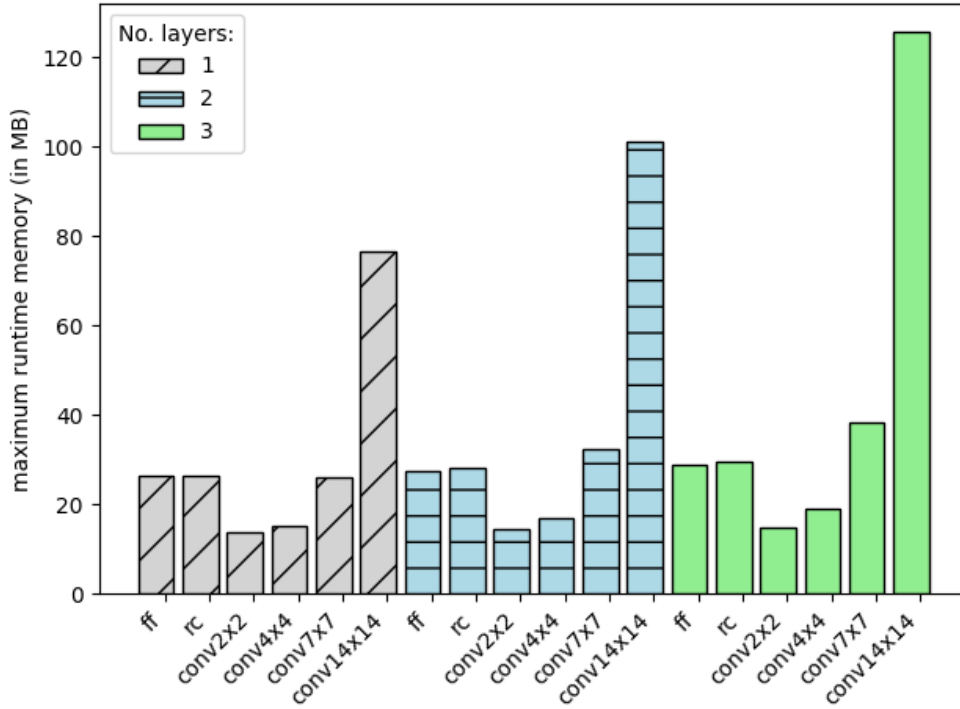


Figure 6.4: batch size of 256; GPU Memory (using torch.cuda.max_memory_allocated)

- *Energy and power:* With an ideal neuromorphic hardware, the CONV-SNN architecture requires on average less energy than other architectures, while it requires more power. Note that this is under the assumption that the SOPs reflect energy and the number of steps reflecting latency. However, as will be shown in a next subsection, this assumption is not valid in the case of using a hardware system of limited resources, such as a CPU with a hierarchical memory system.

6.3.2 Maximum Runtime Memory

The FF-SNN, RC-SNN and CONV-SNN were evaluated for their maximum runtime memory when running on von Neumann architecture (running on GPU). Generally, the maximum run-time memory reflects the number of computing and memory resources needed depending on the host hardware. The maximum runtime memory was measured during inference with a batch size of 256 in PyTorch using PyTorch memory allocation method.

Fig. 6.4 shows the tested SNNs. With increasing network depth, all architectures have an increasing maximum runtime memory. The RC-SNN has slightly higher runtime memory compared to the FF-SNN. Even though the RC-SNN has a doubled num-

Table 6.2: Hardware specifications

CPU	
Model	Intel Xeon Gold 6252N
Clock Speed	2.3 GHz
Memory	L1d: 32KB L1i: 32KB L2 Cache: 1024KB L3 Cache: 36608KB
GPU	
Model	NVIDIA Tesla V100-PCIE-16GB
CUDA Cores	5120 (80 SM)
Clock Speed	135 MHz
Memory	VRAM (global): 16GB L2 cache: 96KB SM shared memory: 6144KB

ber of parameters than of the FF-SNN, the size of the intermediate results dominates compared to the size of the parameters (note that we use an input batch size of 256).

For the CONV-SNN, the maximum runtime memory depends on the size of input patch. When the patch size is smaller, the intermediate results generated by the convolution kernels are smaller, and hence the maximum runtime memory is smaller. It can also be seen that the increase in the number of layers affects less the runtime memory compared to the increase in the patch size. For example, increasing the patch size from 7x7 to 14x14 for the 3 layers architecture increases the maximum runtime memory by 80MB, while increasing the number of layers from 2 to 3 for the patch size 14x14 increases the maximum runtime memory by 20MB.

In comparison to the FF-SNN and RC-SNN, the CONV-SNN performs better in terms of the runtime memory with lower patch sizes. With a patch sizes of 2x2 and 4x4, the CONV-SNN requires approximately half of the runtime memory required for the FF-SNN or the RC-SNN architecture. On the other hand, the CONV-SNN with 14x14 patch size requires around 4x more runtime memory than the FF-SNN or the RC-SNN architectures.

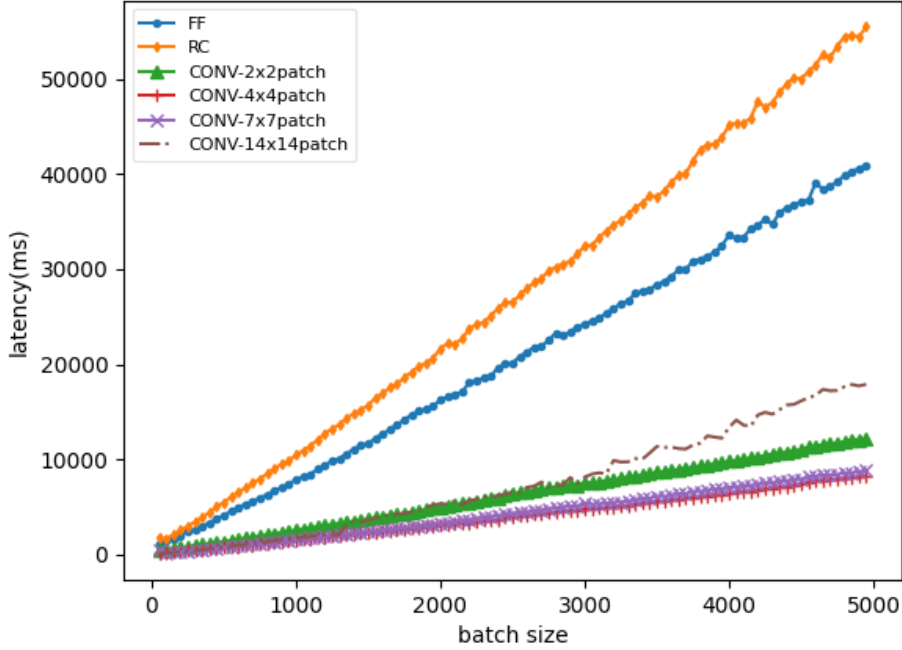


Figure 6.5: Latency for different batch sizes (to emulate different memory resources). This was measured for 3 layers networks processed by a CPU.

6.3.3 Latency and Throughput

The processing efficiency of the different sequential architectures was tested on the CPU and GPU based on PyTorch; table 6.2 presents the specifications of these two hardware systems. The CPU is important to understand the processing efficiency of the network architecture under minimum number of processing units, while the GPU is important to understand the processing efficiency when many processing units cores are available. The purpose of experimenting with CPU and GPU is to evaluate the different sequential architectures under realistic conditions, where the SNN has more neurons and parameters than the number of processing and memory units of the host hardware (e.g., when the SNN has more neurons than the physical neurons in the hardware). This condition forces queuing the neural computations in memory and sequentially processing them. This makes the utilization of a hierarchical memory system dependent on the SNN architecture, which also affects the overall processing efficiency.

To evaluate the network architectures under varying hardware memory, we process input data in a batched way. As we have fixed hardware architectures (e.g. a CPU

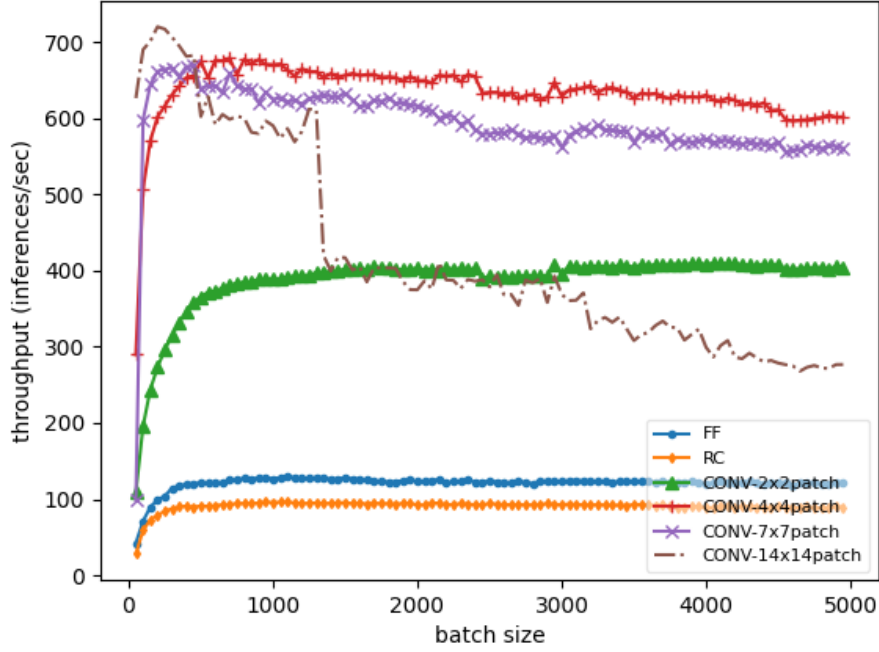


Figure 6.6: Throughput (inferences per second) for different batch sizes. This was measured for 3 layers networks processed by a CPU.

with a fixed size of cache memory), we emulate the change in the hardware memory size by changing the size of the input batch; i.e., if the batch size is 256, 256 outputs are generated at every layer, which increases the memory bottleneck, and hence producing a similar effect decreasing when decreasing the size of the hardware memory. Next, we present the inference latency and throughput (number of inferences per second) for different batch sizes. All SNN architectures were tested with depth of three layers.

Latency on CPU. Fig. 6.5 shows the inference latency of FF-SNN, RC-SNN and CONV-SNN on CPU. For all batch sizes up to 5000, the CONV-SNNs are the lowest in latency, then the FF-SNN, then the RC-SNN. This was expected as the CONV-SNNs have fewer operations (SOPs) compared to the other network architectures. On the other hand, the number of operations is not always proportional to the latency as can be seen from the CONV-SNN architectures of different patch sizes. Notably, two key observations appear in Fig. 6.5: First, under specific patch size, the size of the patch is not proportional to the latency. For example, under batch size 400, the CONV-4x4 is the fastest, then CONV-7x7, then CONV-2x2, then CONV-14x14. Second, the relative latency of the CONV-SNNs change for different batch size. This shows different patch size, fits differently in different hardware memory, resulting in different latency. A

clearer view of thesis can be seen next in the throughput results.

Throughput on CPU. Fig. 6.6 shows the throughput of FF-SNN, RC-SNN and CONV-SNN on CPU. Similar to the latency, the CONV-SNNs have higher throughput than other architectures. With increasing batch size, the throughput of CONV-2x2, FF-SNN and RC-SNN stays almost constant, while it decreases for CONV-4x4, CONV-7x7 and CONV-14x14. The larger the patch size of the CONV-SNN, the faster the decrease in throughput with increasing batch size. For low batch sizes, the CONV-SNN with large batch size have higher throughput. This means that when the memory resources are large enough compared to the runtime memory, the network architecture with the largest batch size performs the best in terms of throughput. With a larger batch size, the throughput changes depend on the patch and batch sizes. For example, at batch size 4000, CONV-4x4 has the highest throughput, then CONV-7x7, then CONV-2x2, then CONV-14x14. From the decreasing trend of the throughput, it is expected that at very large batch sizes, the CONV-SNN with the lowest patch size will have the highest throughput (opposite to the throughput order at low batch sizes). This shows that the processing efficiency of the sequential CONV-SNN depends on the patch size and the size of the intermediate memory in a hierarchical memory system.

Latency on GPU. Fig. 6.7 shows the inference latency of FF-SNN, RC-SNN and CONV-SNN on GPU. The latencies of all networks slightly change with changing the batch size. This is because the GPU processes the inputs in parallel. Similar to the latency on CPU, the CONV-SNN performs better compared to both the RC-SNN and FF-SNN. The RC-SNN has almost the same latency as the FF-SNN, with being slightly slower at larger batch sizes. Even though RC-SNN has almost doubled number of computations than FF-SNN in every layer, these computations in PyTorch are parallelized across the GPU cores, hence their latency is almost constant for different batch sizes. For the CONV-SNN, with small batch sizes, the latency is proportional to the patch size; the larger the patch, the lower the lower the latency. This trend changes with increasing the batch size, where the effect on the utilization of hierarchical memory starts to dominate. More details can be observed next in the throughput results.

Throughput on GPU. Fig. 6.8 shows the throughput of FF-SNN, RC-SNN and CONV-SNN on GPU. For most network architectures, the throughput increases with increasing batch size. This is different from the throughput curve in the case of the CPU, where the throughput in the CPU decreases with increasing batch size. In GPU, the throughput increases as a result of parallelism, while it decreases at some point when the batch size is large compared to the hardware memory. For example, in the case of the CONV-14x14, the throughput starts to decrease around the batch size of 2000. In

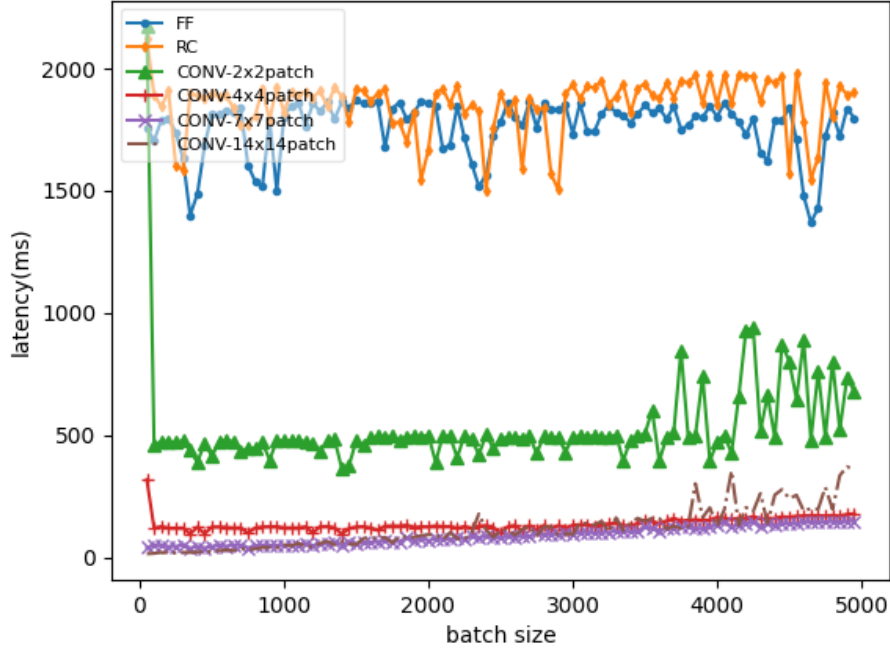


Figure 6.7: Latency for different batch sizes. This was measured for 3 layers networks processed by a GPU.

other words, there is a trade-off (contention effect) between parallelism and memory hierarchy utilization. However, this curve can also be viewed as the throughput curve in the CPU, with the point of decreasing throughput being shifted to the right. Therefore, this implies that a decrease in throughput for a specific network architecture due to inefficient use of the hierarchical memory system can be compensated by parallelism.

Recall: Q3. What is the quantified gain in terms of processing efficiency when reducing the memory overhead of an SNN?

This chapter proposes a solution that processing of an image as a sequence of 2D patches to utilize the dynamicity of the SNN and hence reducing the memory overhead. The proposed solution reduces the runtime memory requirement for convolutional SNNs by at least 4x. It also improves the throughput by at least 2x under a hardware of constrained memory resources.

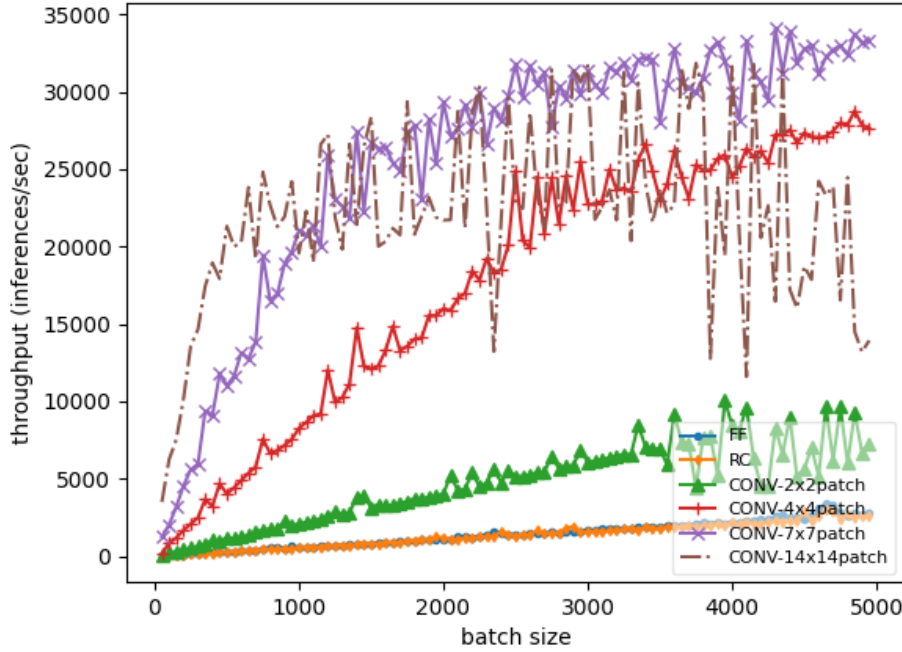


Figure 6.8: Throughput (inferences per second) for different batch sizes. This was measured for 3 layers networks processed by a GPU.

6.4 Limitations

The following presents some open challenges and future work related to this chapter:

- **Training runtime memory:** The sequential processing of images increases the number of steps, and hence the number of iterative execution of the SNN. For example, if we have 49 steps, the SNN needs to be executed 49 times iteratively. When training using BPTT, this increases the size of the training runtime memory, where every iteration builds a new computational graph for the network. This creates a huge memory overhead when training CONV-SNN with long sequences (e.g., when sequentially patching high-resolution images).
- **Training speed:** Another challenge of processing images sequentially is the training speed. During training, large computing and memory resources are usually available. Patching an input image and applying it sequentially limits the parallelism. New training approaches are required to parallelize the computations of the proposed CONV-SNN.
- **Utilizing biologically-plausibility:** This chapter shows the importance of the

dynamical complexity of the spiking neuron in improving the processing efficiency of SNNs. In neuromorphic computing community, simple models (such as LIF) are used when building large networks due to their simpler computational complexity. However, utilizing the rich temporal dynamics of more biologically-plausible models mitigates this computational complexity and even improves the overall processing efficiency. Therefore, extending the proposed approach to new applications and proposing new approaches that utilize the temporal dynamics of the spiking neuron are recommended.

- **Comparing with conventional ANNs:** The proposed architecture utilizes the local reccurancy and dynamics of the SNN. This makes the proposed approach unique to SNNs, which make the sequential image classification a good application. In deep learning, sequential processing of images rely on spatial architectures (e.g., architecture that uses self-attention mechanisms or globally recurrent connections), which requires a large memory overhead. Therefore, it is important to compare this work with approaches from conventional deep learning.
- **More complex datasets and deeper networks:** This chapter tests the proposed ideas on simple images and datasets. Investigating the potential of AdExLIF and BRF on more complex images are also important.

6.5 Conclusion

This chapter proposed the spatiotemporal CONV-SNN. This architecture utilizes the temporal dynamics of the spiking neuron to process images in a patched way. The proposed CON-SNN showed an outperforming results compared to SOTA SNN architectures in terms of processing efficiency and hardware cost. In addition, the proposed network architecture allows for a flexible input patching, which facilitates the design of networks that fit hardware of different memory resources. The proposed approach initiates a new way of processing images using SNN, which can be utilized in specific unexplored applications. Further research is motivated to compare the spatiotemporal CONV-SNN with other networks that process images in a patched way, to use more complex spiking neuron models, and to improve the training speed and memory requirement.

Chapter 7

Conclusion

This dissertation investigates the problem of the memory overhead in SNNs, with the aim of increasing the processing efficiency of SNNs and reducing their hardware cost.

This dissertation contributes to the SNNs as follows: As the problem of the SNNs' memory overhead is neglected in the literature, chapter 2 discussed and described the problem in detail. It also answers Q2 by proposing two approaches to address the memory overhead in the SNNs; reducing the number of active neurons and utilizing the dynamicity. In chapter 3, the key design principle is set (i.e., designing narrow dataflow SNNs) and reused throughout the whole dissertation (answer to Q1). Chapter 3 also proposes task specialization, a solution to reduce the memory overhead by dividing the AI classification task implemented by the SNN into multiple sub-tasks. This approach reduces the memory overhead by $\tilde{10}x$ in terms of energy and latency as a result of reducing the number of memory operations and increasing the utilization of the memory hierarchy (answer to Q3). This solution has three main limitations: (1) degradation of accuracy in some tasks, (2) the training burden and (3) the expertise requirement to design the sub-tasks. Therefore, in chapter 4, the per layer specialization was proposed to address these challenges. In the per layer specialization, the layers were specialized automatically during training by the MoE architecture. The per layer specialization improves the processing efficiency by 3x in terms of latency as a result of reducing the memory requirements, reducing the number of memory operations and increasing the utilization of hierarchical memory (answer to Q3). These two solutions of specialization reduce the memory overhead by adopting the approach of reducing the number of active neurons. In chapters 5 and 6, a new approach is adopted to reduce the memory overhead by utilizing the intrinsic temporal dynamics of the spiking neuron. In chapter 5, the capability of the spiking neuron models for extracting features temporally were

investigated. The use of more biologically-plausible neurons, such as AdExLIF, was proposed to temporally extract features from images. This shows that the dynamical properties of the SNN can be utilized to reduce the memory overhead rather than only causing it (answer to Q4). In chapter 6, these biologically-plausible neuron models were used in a convolutional SNN to process images as a sequence of 2D patches (answer to Q5). This introduces a flexibility in the design of SNNs, which allows them to be efficiently mapped in hardware platforms of different memory resources. This allows for improvement in the processing efficiency as well as in the memory requirements; the runtime memory requirement of the CONV-SNN is improved by at least 4x and the throughput is improved by at least 2x (answer to Q3).

The proposed solutions advance the processing efficiency of the neuromorphic systems from the neural network level. The achieved results are motivated by or based on the assumption that the host hardware of the SNN is limited in terms of memory resources. This assumption was made as the software and algorithmic developments are faster than the hardware developments, and hence the scaling of SNNs are expected to be faster than the scaling of the neuromorphic hardware. Therefore, we assume that the memory resources in the neuromorphic hardware will be always limited compared to the SNN memory requirements. On the other hand, this assumption fails if the design of the neuromorphic hardware is customized to the SNN. In addition, the research methodology of proposed work in this dissertation is based on testing the memory overhead on conventional hardware (such as CPU or GPU). As the neuromorphic hardware is usually based on non-conventional hardware (such as the analog neuromorphic chips), the presented processing efficiency gains may not necessarily reflect the exact gains in different hardware architectures.

In summary, this dissertation contributes to the processing efficiency of the spiking neural networks by reducing their memory overhead. It proposed fundamental solutions that utilize existing approaches in ANNs as well as solutions that are specific for SNNs. More research is required to test the proposed solutions on different types neuromorphic hardware. In addition, more research is motivated towards the applicability of the SNN in a sequenced (tokenized) images or signals.

7.1 Future Work

This dissertation opens up research space for several challenges and future work. First, studying the proposed approaches in the context of the available neuromorphic hardware is required. For example, the temporal processing using the AdExLIF neuron may

require precise values for the neuron parameters (for a very sharp notch filter, we may require an adaptation constant of 0.99). This high precision cannot be valid in some analog neuromorphic hardware such as TEXEL [33] and DYNAP [88]. Therefore, design constraints and requirements can be defined for such hardware for enabling applications and networks similar to those presented in this dissertation.

Second, training an SNN using BPTT when temporally processing images and signals can be extremely time consuming and memory intensive. This puts the SNN far beyond the conventional ANNs when it comes for deployment in complex applications that require large scale SNNs. Therefore, there is a practical need for improving the training efficiency of the SNNs, especially for SNN training that requires long input sequences.

Third, an exploration of the spiking neuron models is required. The mathematical modeling of the spiking neuron is important to enrich its temporal dynamics while maintaining a computationally simple model. In addition, the spiking neuron model can be modified to fit some hardware constraints without affecting their functionality. Therefore, a deeper study of the spiking neuron model under different functional and computational constraints is required. This dissertation tests the proposed approaches on simple applications (e.g., image classification on MNIST and CIFAR). Investigating the potential of AdExLIF and BRF on more complex datasets and applications are also important.

This dissertation opens the ground for a new set of SNN applications that require a sequential processing of 2D images. For example, some applications may require low cost hardware cameras that can be used to capture an image (a target visual scene) as sequence of 2D patches. This work allows for similar applications to be deployed on some of the proposed SNN architectures.

Finally, the integration of the proposed solutions was also left for future work. In fact, the proposed two approaches are orthogonal to each other. We expect this integration to further improve the memory requirement and the processing efficiency of the SNNs. For example, the early layers in the per-layer task specialization solution can be divided into more experts when applying the input image as a sequence of 2D patches; i.e., a 2D patch contains less features compared to a full image, which requires less neurons per expert. The integration of task specialization and the spatiotemporal convolution solutions is also expected to improve the overall processing efficiency. The task specialization reduces the latency by at least 10x by eliminating the dependencies between different sub-SNNs. Applying the spatiotemporal convolution allows a further reduction in every sub-SNN by up to 3x. Therefore, the expected improvement when

integrating these two solutions is 30x. This shows the importance of extending the work in this dissertation to integrate the proposed solutions.

7.2 List of Publications

The following shows the list of articles published/prepared from this PhD dissertation:

- **From Chapter 3:** Muath Abu Lebdeh, et al. "Efficient processing of spiking neural networks via task specialization", IEEE Transactions on Emerging Topics in Computational Intelligence, 8(5):3603–3613, 2024.
- **From Chapter 4:** Muath Abu Lebdeh el al. "Per layer specialization for memory-efficient spiking neural networks," Proceedings of the International Conference on Neuromorphic Systems (ICONS), pages 169–176, 2024.
- **From Chapters 5 and 6:** Muath Abu Lebdeh el al. "Memory-efficient spatiotemporal convolutional spiking neural networks for vision applications," article submitted to the IEEE Transactions in Emerging Topics in Computational Intelligence.

Bibliography

- [1] Khaled Aboumerhi, Amparo Güemes, Hongtao Liu, Francesco Tenore, and Ralph Etienne-Cummings. Neuromorphic applications in medicine. *Journal of Neural Engineering*, 20(4):041004, 2023.
- [2] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015.
- [3] Badr AlKhamissi, Muhammad ElNokrashy, and David Bernal-Casas. Deep spiking neural networks with resonate-and-fire neurons. *arXiv preprint arXiv:2109.08234*, 2021.
- [4] Arnon Amir, Pallab Datta, William P Risk, Andrew S Cassidy, Jeffrey A Kusnitz, Steve K Esser, Alexander Andreopoulos, Theodore M Wong, Myron Flickner, Rodrigo Alvarez-Icaza, et al. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE, 2013.
- [5] 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.
- [6] Malyaban Bal and Abhronil Sengupta. Spikingbert: Distilling bert to train spiking language models using implicit differentiation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 10998–11006, 2024.

- [7] David Beniaguev, Sapir Shapira, Idan Segev, and Michael London. Multiple synaptic contacts combined with dendritic filtering enhance spatio-temporal pattern recognition capabilities of single neurons. *bioRxiv*, pages 2022–01, 2022.
- [8] Olivier Bichler, Damien Querlioz, Simon J Thorpe, Jean-Philippe Bourgoin, and Christian Gamrat. Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity. *Neural networks*, 32:339–348, 2012.
- [9] Sander M Bohte, Joost N Kok, and Johannes A La Poutr . Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, volume 48, pages 419–424. Bruges, 2000.
- [10] Irem Boybat, Manuel Le Gallo, SR Nandakumar, Timoleon Moraitis, Thomas Parnell, Tomas Tuma, Bipin Rajendran, Yusuf Leblebici, Abu Sebastian, and Evangelos Eleftheriou. Neuromorphic computing with multi-memristive synapses. *Nature communications*, 9(1):2514, 2018.
- [11] 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.
- [12] Natalia Caporale and Yang Dan. Spike timing–dependent plasticity: a hebbian learning rule. *Annu. Rev. Neurosci.*, 31(1):25–46, 2008.
- [13] Biswadeep Chakraborty, Beomseok Kang, Harshit Kumar, and Saibal Mukhopadhyay. Sparse spiking neural network: Exploiting heterogeneity in timescales for pruning recurrent snn. In *The Twelfth International Conference on Learning Representations*, 2024.
- [14] Zitian Chen, Yikang Shen, Mingyu Ding, Zhenfang Chen, Hengshuang Zhao, Erik G Learned-Miller, and Chuang Gan. Mod-squad: Designing mixtures of experts as modular multi-task learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11828–11837, 2023.
- [15] Kyunghyun Cho, B van Merri nboer, Caglar Gulcehre, F Bougares, H Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

-
- [16] Junyoung Chung, Kyunghyun Cho, and Yoshua Bengio. A character-level decoder without explicit segmentation for neural machine translation. In *54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*, pages 1693–1703. Association for Computational Linguistics (ACL), 2016.
- [17] Intel Corporation. Intel builds world’s largest neuromorphic system, 2024. Accessed: Mar. 10, 2025.
- [18] Bart Cox, Robert Birke, and Lydia Y Chen. Memory-aware and context-aware multi-dnn inference on the edge. *Pervasive and Mobile Computing*, 83:101594, 2022.
- [19] Gourav Datta, Haoqin Deng, Robert Aviles, and Peter A Beerel. Towards energy-efficient, low-latency and accurate spiking lstms. *arXiv preprint arXiv:2210.12613*, 2022.
- [20] Simon Davidson and Steve B Furber. Comparison of artificial and spiking neural networks on digital hardware. *Frontiers in Neurosc.*, 15:651141, 2021.
- [21] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.
- [22] Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.
- [23] Juan P Dominguez-Morales, Qian Liu, Robert James, Daniel Gutierrez-Galan, Angel Jimenez-Fernandez, Simon Davidson, and Steve Furber. Deep spiking neural network model for time-variant signals classification: a real-time speech recognition approach. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.
- [24] Daniel Elbrecht, Shruti R Kulkarni, Maryam Parsa, J Parker Mitchell, and Catherine D Schuman. Evolving ensembles of spiking neural networks for neuromorphic systems. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1989–1994. IEEE, 2020.

- [25] Jason K Eshraghian. snntorch documentation, 2021.
- [26] Jason K Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D Lu. Training spiking neural networks using lessons from deep learning. *arXiv preprint arXiv:2109.12894*, 2021.
- [27] Haowen Fang, Amar Shrestha, Ziyi Zhao, and Qinru Qiu. Exploiting neuron and synapse filter dynamics in spatial temporal learning of deep spiking neural network. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 2799–2806, 2021.
- [28] Clayton Fields and Casey Kennington. Vision language transformers: A survey. *arXiv preprint arXiv:2307.03254*, 2023.
- [29] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.
- [30] Wulfram Gerstner and Werner M Kistler. Mathematical formulations of hebbian learning. *Biological cybernetics*, 87(5):404–415, 2002.
- [31] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213, 2019.
- [32] Hector A Gonzalez, Jiaxin Huang, Florian Kelber, Khaleelulla Khan Nazeer, Tim Langer, Chen Liu, Matthias Lohrmann, Amirhossein Rostami, Mark Schöne, Bernhard Vogginger, et al. Spinnaker2: A large-scale neuromorphic system for event-based and asynchronous machine learning. *arXiv preprint arXiv:2401.04491*, 2024.
- [33] Hugh Greatorex, Ole Richter, Michele Mastella, Madison Cotteret, Philipp Klein, Maxime Fabre, Arianna Rubino, Willian Soares Girão, Junren Chen, Martin Ziegler, et al. Texel: A neuromorphic processor with on-chip learning for beyond-cmos device integration. *arXiv preprint arXiv:2410.15854*, 2024.
- [34] Yufei Guo, Yuanpei Chen, Liwen Zhang, Xiaode Liu, Yinglei Wang, Xuhui Huang, and Zhe Ma. Im-loss: information maximization loss for spiking neural networks. *Advances in Neural Information Processing Systems*, 35:156–166, 2022.

- [35] Md Sakib Hasan, Catherine D Schuman, Zhongyang Zhang, Tauhidur Rahman, and Garrett S Rose. Spike-based neuromorphic computing for next-generation computer vision. In *Computer Vision*, pages 312–339. Chapman and Hall/CRC.
- [36] Saya Higuchi, Sebastian Kairat, Sander Bohte, and Sebastian Otte. Balanced resonate-and-fire neurons. In *International Conference on Machine Learning*, pages 18305–18323. PMLR, 2024.
- [37] Michael Hopkins, Garibaldi Pineda-Garcia, Petruț A Bogdan, and Steve B Furber. Spiking neural networks for computer vision. *Interface focus*, 8(4):20180007, 2018.
- [38] Yifan Hu, Lei Deng, Yujie Wu, Man Yao, and Guoqi Li. Advancing spiking neural networks toward deep residual learning. *IEEE Trans. Neural Netw. Learn. Syst.*, 2024.
- [39] Eric Hunsberger and Chris Eliasmith. Spiking deep networks with LIF neurons. *arXiv*, 2015.
- [40] Giacomo Indiveri, Fabio Stefanini, and Elisabetta Chicca. Spike-based learning with a generalized integrate and fire silicon neuron. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1951–1954. IEEE, 2010.
- [41] Eugene M Izhikevich. Resonate-and-fire neurons. *Neural networks*, 14(6-7):883–894, 2001.
- [42] Angel Jiménez-Fernández, Elena Cerezuela-Escudero, Lourdes Miró-Amarante, Manuel Jesus Domínguez-Morales, Francisco de Asís Gómez-Rodríguez, Alejandro Linares-Barranco, and Gabriel Jiménez-Moreno. A binaural neuromorphic auditory sensor for fpga: a spike signal processing approach. *IEEE transactions on neural networks and learning systems*, 28(4):804–818, 2016.
- [43] Saeed Kargar and Faisal Nawab. Challenges and future directions for energy, latency, and lifetime improvements in nvms. *Distributed and Parallel Databases*, pages 1–27, 2022.
- [44] Kento Kawaharazuka, Tatsuya Matsushima, Andrew Gambardella, Jiaxian Guo, Chris Paxton, and Andy Zeng. Real-world robot applications of foundation models: A review. *Advanced Robotics*, 38(18):1232–1254, 2024.

- [45] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.
- [46] Eunsu Kim and Youngmin Kim. Exploring the potential of spiking neural networks in biomedical applications: advantages, limitations, and future perspectives. *Biomedical Engineering Letters*, 14(5):967–980, 2024.
- [47] Sangyeob Kim, Sangjin Kim, Soyeon Um, Soyeon Kim, Juhyoung Lee, and Hoi-Jun Yoo. Snpu: An energy-efficient spike domain deep-neural-network processor with two-step spike encoding and shift-and-accumulation unit. *IEEE Journal of Solid-State Circuits*, 58(10):2812–2825, 2023.
- [48] Youngeun Kim, Yuhang Li, Abhishek Moitra, Ruokai Yin, and Priyadarshini Panda. Sharing leaky-integrate-and-fire neurons for memory-efficient spiking neural networks. *Frontiers in Neurosci.*, 17, 2023.
- [49] Youngeun Kim, Yuhang Li, Hyoungeob Park, Yeshwanth Venkatesha, Anna Hambitzer, and Priyadarshini Panda. Exploring temporal information dynamics in spiking neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 8308–8316, 2023.
- [50] Youngeun Kim, Yuhang Li, Hyoungeob Park, Yeshwanth Venkatesha, and Priyadarshini Panda. Neural architecture search for spiking neural networks. In *European conference on computer vision*, pages 36–56. Springer, 2022.
- [51] Youngeun Kim, Yuhang Li, Hyoungeob Park, Yeshwanth Venkatesha, Ruokai Yin, and Priyadarshini Panda. Exploring lottery ticket hypothesis in spiking neural networks. In *ECCV*, pages 102–120. Springer, 2022.
- [52] Youngeun Kim and Priyadarshini Panda. Optimizing deeper spiking neural networks for dynamic vision sensing. *Neural Networks*, 144:686–698, 2021.
- [53] Youngeun Kim, Hyoungeob Park, Abhishek Moitra, Abhiroop Bhattacharjee, Yeshwanth Venkatesha, and Priyadarshini Panda. Rate coding or direct coding: Which one is better for accurate, robust, and energy-efficient spiking neural networks? In *ICASSP*, pages 71–75. IEEE, 2022.
- [54] Suhas Kumar, R Stanley Williams, and Ziwen Wang. Third-order nanocircuit elements for neuromorphic engineering. *Nature*, 585(7826):518–523, 2020.

-
- [55] Souvik Kundu, Massoud Pedram, and Peter A Beerel. Hire-snn: Harnessing the inherent robustness of energy-efficient deep spiking neural networks by training with crafted input noise. In *IEEE/CVF ICCV*, pages 5209–5218, 2021.
 - [56] Muath Abu Lebdeh, Kasim Sinan Yildirim, and Davide Brunelli. Efficient processing of spiking neural networks via task specialization. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 8(5):3603–3613, 2024.
 - [57] Muath Abu Lebdeh, Kasim Sinan Yildirim, and Davide Brunelli. Per layer specialization for memory-efficient spiking neural networks. In *2024 International Conference on Neuromorphic Systems (ICONS)*, pages 169–176. IEEE, 2024.
 - [58] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in neuroscience*, 10:508, 2016.
 - [59] Alexander J Leigh, Moslem Heidarpur, and Mitra Mirhassani. Selective input sparsity in spiking neural networks for pattern classification. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 799–803. IEEE, 2022.
 - [60] Yang Li and Yi Zeng. Efficient and accurate conversion of spiking neural network with burst spikes. *arXiv preprint arXiv:2204.13271*, 2022.
 - [61] Yuhang Li, Tamar Geller, Youngeun Kim, and Priyadarshini Panda. SEENN: Towards temporal spiking early exit neural networks. In *NeurIPS*, 2023.
 - [62] Yuhang Li, Abhishek Moitra, Tamar Geller, and Priyadarshini Panda. Input-aware dynamic timestep spiking neural networks for efficient in-memory computing. *arXiv*, 2023.
 - [63] Chit-Kwan Lin, Andreas Wild, Gautham N Chinya, Tsung-Han Lin, Mike Davies, and Hong Wang. Mapping spiking neural networks onto a manycore neuromorphic architecture. *ACM SIGPLAN Notices*, 53(4):78–89, 2018.
 - [64] Zeyu Liu, Gourav Datta, Anni Li, and Peter Anthony Beerel. LMUFormer: Low complexity yet powerful spiking model with legendre memory units. In *The Twelfth International Conference on Learning Representations*, 2024.
 - [65] Michele Mastella, Hugh Greatorex, Madison Cotteret, Ella Janotte, Willian Soares Girao, Ole Richter, and Elisabetta Chicca. Synaptic normalisation for

- on-chip learning in analog cmos spiking neural networks. In *Proceedings of the 2023 International Conference on Neuromorphic Systems*, pages 1–4, 2023.
- [66] Michele Mastella, Tesse Tiemens, and Elisabetta Chicca. Texture recognition using a biologically plausible spiking phase-locked loop model for spike train frequency decomposition. *arXiv preprint arXiv:2403.09723*, 2024.
- [67] 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.
- [68] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *2015 IEEE workshop on automatic speech recognition and understanding (ASRU)*, pages 167–174. IEEE, 2015.
- [69] Vitalii Mikheev, Anastasia Chouprik, Yury Lebedinskii, Sergei Zarubin, Yury Matveyev, Ekaterina Kondratyuk, Maxim G Kozodaev, Andrey M Markeev, Andrei Zenkevich, and Dmitrii Negrov. Ferroelectric second-order memristor. *ACS applied materials & interfaces*, 11(35):32108–32114, 2019.
- [70] John Moon, Wen Ma, Jong Hoon Shin, Fuxi Cai, Chao Du, Seung Hwan Lee, and Wei D Lu. Temporal data classification and forecasting using a memristor-based reservoir computing system. *Nature Electronics*, 2(10):480–487, 2019.
- [71] 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*, 12(1):106–122, 2017.
- [72] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? *Advances in neural information processing systems*, 32, 2019.
- [73] Byunggook Na, Jisoo Mok, Seongsik Park, Dongjin Lee, Hyeokjun Choe, and Sungroh Yoon. Autosnn: towards energy-efficient spiking neural networks. In *ICML*, pages 16253–16269. PMLR, 2022.
- [74] Joseph S Najem, Graham J Taylor, Ryan J Weiss, Md Sakib Hasan, Garrett Rose, Catherine D Schuman, Alex Belianinov, C Patrick Collier, and Stephen A Sarles. Memristive ion channel-doped biomembranes as synaptic mimics. *ACS nano*, 12(5):4702–4711, 2018.

-
- [75] Georgiana Neculae, Oliver Rhodes, and Gavin Brown. Ensembles of spiking neural networks. *arXiv*, 2020.
- [76] 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*, 36(6):51–63, 2019.
- [77] Garrick Orchard, E Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T Sommer, and Mike Davies. Efficient neuromorphic signal processing with loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 254–259. IEEE, 2021.
- [78] Antardipan Pal, Yong Zhang, and Dennis D Yau. Monolithic and single-functional-unit level integration of electronic and photonic elements: Fet-let hybrid 6t sram. *Photonics Research*, 9(7):1369–1378, 2021.
- [79] 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. *ACM SIGARCH computer architecture news*, 45(2):27–40, 2017.
- [80] Chandrashekhar S Patil, Sourabh B Ghode, Jungmin Kim, Girish Kamble, Somnath Kundale, Abdul Mannan, Yongbin Ko, Muhammad Noman, Qazi Muhammad Saqib, Swapnil Patil, et al. Neuromorphic devices for electronic skin applications. *Materials Horizons*, 2025.
- [81] Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. The brainscales-2 accelerated neuromorphic system with hybrid plasticity. *Frontiers in Neuroscience*, 16:795876, 2022.
- [82] Nicolas Perez-Nieves and Dan Goodman. Sparse spiking gradient descent. *Advances in Neural Information Processing Systems*, 34:11795–11808, 2021.
- [83] Ewa Piatkowska, Ahmed Nabil Belbachir, Stephan Schraml, and Margrit Gelautz. Spatiotemporal multiple persons tracking using dynamic vision sensor. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 35–40. IEEE, 2012.

- [84] 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. IEEE, 2021.
- [85] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. Mantis: enabling energy-efficient autonomous mobile agents with spiking neural networks. In *ICARA*, pages 197–201. IEEE, 2023.
- [86] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.
- [87] Nitin Rathi and Kaushik Roy. Stdp based unsupervised multimodal learning with cross-modal processing in spiking neural networks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(1):143–153, 2018.
- [88] Ole Richter, Chenxi Wu, Adrian M Whatley, German Köstinger, Carsten Nielsen, Ning Qiao, and Giacomo Indiveri. Dynap-se2: a scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor. *NCE*, 4(1):014003, 2024.
- [89] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *Advances in Neural Information Processing Systems*, 34:8583–8595, 2021.
- [90] Alberto Rodriguez-Fernandez, Carlo Cagli, Luca Perniola, Enrique Miranda, and Jordi Suñé. Characterization of hfo2-based devices with indication of second order memristor effects. *Microelectronic Engineering*, 195:101–106, 2018.
- [91] Bodo Rueckauer, Connor Bybee, Ralf Goettsche, Yashwardhan Singh, Joyesh Mishra, and Andreas Wild. Nxtf: An api and compiler for deep spiking neural networks on intel loihi. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(3):1–22, 2022.
- [92] Ali Safa, Jonah Van Assche, Mark Daniel Alea, Francky Catthoor, and Georges GE Gielen. Neuromorphic near-sensor computing: from event-based sensing to edge learning. *Ieee Micro*, 42(6):88–95, 2022.

-
- [93] Ali Safa, Tim Verbelen, Ilja Ocket, André Bourdoux, Hichem Sahli, Francky Catthoor, and Georges Gielen. Fusing event-based camera and radar for slam using spiking neural networks with continual stdp learning. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2782–2788. IEEE, 2023.
- [94] Sai Sanjeet, Rahul K Meena, Bibhu Datta Sahoo, Keshab K Parhi, and Masahiro Fujita. Iir filter-based spiking neural network. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2023.
- [95] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13:95, 2019.
- [96] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv*, 2017.
- [97] Xueyuan She, Saurabh Dash, and Saibal Mukhopadhyay. Sequence approximation using feedforward spiking neural network for spatiotemporal learning: Theory and optimization methods. In *International Conference on Learning Representations*, 2021.
- [98] Guobin Shen, Dongcheng Zhao, and Yi Zeng. Exploiting high performance spiking neural networks with efficient spiking patterns. *CoRR*, abs/2301.12356, 2023.
- [99] Jiangrong Shen, Yu Zhao, Jian K. Liu, and Yueming Wang. Hybridsnn: Combining bio-machine strengths by boosting adaptive spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 34(9):5841–5855, 2023.
- [100] Xinyu Shi, Jianhao Ding, Zecheng Hao, and Zhaoifei Yu. Towards energy efficient spiking neural networks: An unstructured pruning framework. In *The Twelfth International Conference on Learning Representations*, 2024.
- [101] Ravid Shwartz-Ziv and Naftali Tishby. Opening the black box of deep neural networks via information. *arXiv*, 2017.
- [102] A software framework for neuromorphic systems. Lava software framework.

- [103] Evangelos Stromatias, Miguel Soto, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data. *Frontiers in neuroscience*, 11:350, 2017.
- [104] Jan Stuijt, Manolis Sifalakis, Amirreza Yousefzadeh, and Federico Corradi. μ brain: An event-driven and fully synthesizable architecture for spiking neural networks. *Frontiers in neuroscience*, 15:538, 2021.
- [105] Kazuma Suetake, Takuya Ushimaru, Ryuji Saiin, and Yoshihide Sawada. Synaptic interaction penalty: Appropriate penalty term for energy-efficient spiking neural networks. *TMLR*, 2024.
- [106] 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, 2017.
- [107] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. *Efficient processing of deep neural networks*. Springer, 2020.
- [108] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
- [109] Gessyca Maria Tovar, Eric Shun Fukuda, Tetsuya Asai, Tetsuya Hirose, and Yoshihito Amemiya. Analog cmos circuits implementing neural segmentation model based on symmetric stdp learning. In *Neural Information Processing: 14th International Conference, ICONIP 2007, Kitakyushu, Japan, November 13-16, 2007, Revised Selected Papers, Part II 14*, pages 117–126. Springer, 2008.
- [110] Anup Vanarse, Josafath Israel Espinosa-Ramos, Adam Osseiran, Alexander Rassau, and Nikola Kasabov. Application of a brain-inspired spiking neural network architecture to odor data classification. *Sensors*, 20(10):2756, 2020.
- [111] M Lakshmi Varshika, Adarsha Balaji, Federico Corradi, Anup Das, Jan Stuijt, and Francky Catthoor. Design of many-core big little μ brains for energy-efficient embedded neuromorphic computing. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1011–1016. IEEE, 2022.
- [112] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. tinysnn: Towards memory-and energy-efficient spiking neural networks. *arXiv*, 2022.

-
- [113] Wenjie Wei, Yu Liang, Ammar Belatreche, Yichen Xiao, Honglin Cao, Zhenbang Ren, Guoqing Wang, Malu Zhang, and Yang Yang. Q-snns: Quantized spiking neural networks. In *Proceedings of the 32nd ACM international conference on multimedia*, pages 8441–8450, 2024.
- [114] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [115] Jibin Wu, Yansong Chua, and Haizhou Li. A biologically plausible speech recognition framework based on spiking neural networks. In *2018 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2018.
- [116] Jibin Wu, Yansong Chua, Malu Zhang, Haizhou Li, and Kay Chen Tan. A spiking neural network framework for robust sound classification. *Frontiers in neuroscience*, 12:836, 2018.
- [117] Jibin Wu, Emre Yilmaz, Malu Zhang, Haizhou Li, and Kay Chen Tan. Deep spiking neural networks for large vocabulary automatic speech recognition. *Frontiers in neuroscience*, 14:199, 2020.
- [118] Rong Xiao, Yu Wan, Baosong Yang, Haibo Zhang, Huajin Tang, Derek F Wong, and Boxing Chen. Towards energy-preserving natural language understanding with spiking neural networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31:439–447, 2022.
- [119] Peng Xu, Xiatian Zhu, and David A Clifton. Multimodal learning with transformers: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(10):12113–12132, 2023.
- [120] Qi Xu, Yaxin Li, Jiangrong Shen, Jian K Liu, Huajin Tang, and Gang Pan. Constructing deep spiking neural networks from artificial neural networks with knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7886–7895, 2023.
- [121] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. Spiking neural networks and their applications: A review. *Brain sciences*, 12(7):863, 2022.
- [122] Helin Yang, Kwok-Yan Lam, Liang Xiao, Zehui Xiong, Hao Hu, Dusit Niyato, and H Vincent Poor. Lead federated neuromorphic learning for wireless edge artificial intelligence. *Nature communications*, 13(1):4269, 2022.

- [123] Ruokai Yin, Abhishek Moitra, Abhiroop Bhattacharjee, Youngeun Kim, and Priyadarshini Panda. Sata: Sparsity-aware training accelerator for spiking neural networks. *IEEE TCAD*, 2022.
- [124] Dehao Zhang, Shuai Wang, Yichen Xiao, Wenjie Wei, Yimeng Shan, Malu Zhang, and Yang Yang. Memory-free and parallel computation for quantized spiking neural networks. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2025.
- [125] Xin Zhang, Liangxiu Han, Tam Sobeih, Lianghao Han, and Darren Dancey. A novel spike transformer network for depth estimation from event cameras via cross-modality knowledge distillation. *arXiv preprint arXiv:2404.17335*, 2024.
- [126] Hanle Zheng, Yujie Wu, Lei Deng, Yifan Hu, and Guoqi Li. Going deeper with directly-trained larger spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11062–11070, 2021.
- [127] Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *International Journal of Machine Learning and Cybernetics*, pages 1–65, 2024.
- [128] Chenlin Zhou, Han Zhang, Zhaokun Zhou, Liutao Yu, Liwei Huang, Xiaopeng Fan, Li Yuan, Zhengyu Ma, Huihui Zhou, and Yonghong Tian. Qkformer: Hierarchical spiking transformer using qk attention. *arXiv preprint arXiv:2403.16552*, 2024.