University of California
Santa Barbara

Towards Efficient and Robust Neuromorphic Computing Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Ling Liang

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Peng Li, Co-Chair
Professor Yufei Ding
Professor Tim Sherwood
Professor Zheng Zhang

September 2022

The Dissertation of Ling Liang is approved.

Professor Yufei Ding

Professor Tim Sherwood

Professor Zheng Zhang

Professor Peng Li, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

April 2022

Towards Efficient and Robust Neuromorphic Computing Systems

by

Ling Liang

Curriculum Vitæ
Ling Liang

Education

2022	Ph.D. in Electrical Computer Engineering (Expected), University of California, Santa Barbara.
2017	M.A. in Electrical Engineering, University of Sourthern California.
2015	B.S. in Electrical Engineering, Beijing University of Posts and Telecommunications.

Publications

[C1]. Jilan Lin*, **Ling Liang***, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, Yuan Xie. "INSPIRE: IN-Storage Private Information REtrieval via Protocol and Architecture Co-design." *The 49th Annual International Symposium on Computer Architecture (ISCA), 2022. (To appear)*

[C2]. **Ling Liang**, Zhaodong Chen, Lei Deng, Fengbin Tu, Guoqi Li, Yuan Xie. "Accelerating Spatiotemporal Supervised Training of Large Scale Spiking Neural Network on GPU." *Design Automation & Test in Europe Conference & Exhibition (DATE), 2022.* (See Chapter 4)

[C3]. Xing Hu, **Ling Liang**, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yuefei Ding, Chang Liu, Tim Sherwood, Yuan Xie. "A DNN Model Extraction Framework Based on Learning Architectural Hints." *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.*

[C4]. Yu Ji, **Ling Liang**, Lei Deng, Youyang Zhang, Youhui Zhang, Yuan Xie. "TETRIS: TilE-matching the TRemendous Irregular Sparsity", *Advances in Neural Information Processing Systems (NeurIPS), 2018.*

[J1]. **Ling Liang**, Zheng Qu, Zhaodong Chen, Fengbin Tu, Yuejie Wu, Lei Deng, Guoqi Li, Peng Li, Yuan Xie. "H2Learn: High-efficiency Learning Accelerator for High-accuracy Spiking Neural Networks." *IEEE Transactions on Computer-Aided Design of integrated Circuits and Systems (TCAD), 2022.* (See Chapter 3)

[J2]. **Ling Liang**, Xing Hu, Lei Deng, Yujie Wu, Guoqi Li, Yufei Ding, Peng Li, Yuan Xie. "Exploring Adversarial Attack in Spiking Neural Networks with Spike-Compatible Gradient." *IEEE Transactions on Neural Networks and Learning Systems (TNNLS), 2021.* (See Chapter 5)

[J3]. **Ling Liang**, Jianyu Xu, Lei Deng, Mingyu Yan, Xing Hu, Zheng Zhang, Guoqi Li, Yuan Xie. "Fast Search of The Optimal Contraction Sequence in Tensor Network." *IEEE Journal of Selected Topics in Signal Processing (JSTSP), 2021.*

[J4]. Jianyu Xu, **Ling Liang**, Lei Deng, Changyun Wen, Yuan Xie, Guoqi Li. “Towards a Polynomial Algorithm for Optimal Contraction Sequence of Tensor Networks From Trees.” *Pysical Review E (PRE), 2019.*

[J5]. Lei Deng*, **Ling Liang***, Guanrui Wang, Liang Chang, Xing Hu, Xin Ma, Liu, Liu, Jing Pei, Guoqi Li, Yuan Xie. “Semimap: A Semi-Folded Convolution Mapping for Speed-Overhead Balance on Crossbars.” *IEEE Transactions on Computer-Aided Design of integrated Circuits and Systems (TCAD), 2018.*

[J6]. **Ling Liang**, Lei Deng, Yueling Zeng, Xing Hu, Yu Ji, Xin Ma, Guoqi Li, Yuan Xie. “Crossbar-aware neural network pruning.” *IEEE Access, 2018.*

Abstract

Towards Efficient and Robust Neuromorphic Computing Systems

by

Ling Liang

Spiking neural networks (SNNs) are known as the third generation of neural networks. For an SNN, the bio-inspired neural dynamics endow the great potential to simulate the neural behaviors of the brain; the additional temporal information propagation provides a larger space to make a comprehensive decision; the binary format and the sparse activities of spikes make SNNs quite energy efficient when considering the real deployment. High accuracy, high efficiency, and high robustness are several attractive features of the brain.

In the early stage, the bio-plausible unsupervised training methods are the mainstream but restrict the learning accuracy of SNNs. Recently, the emerging supervised training algorithms inspired by backpropagation through time (BPTT) have successfully boosted the accuracy. However, the implementation complexity of these BPTT-based algorithms is explosively growing, which raises a much higher demand for hardware resources. To improve the training efficiency, this dissertation proposes two solutions to optimize the BPTT-based training. The first solution is to directly design an ASIC accelerator for SNNs while the other is to optimize the dataflows on GPU.

On the other side, how to improve the robustness of SNNs is critical for building a reliable neuromorphic system. This dissertation first discusses how to disturb an SNN model through adversarial examples, and then conducts an in-depth analysis of the SNN robustness. With the observations, a robust training method for SNNs is inspired by the robustness certification in neural networks.

Contents

Chapter 1

Introduction

Humans never stop trying to understand the mechanism of the brain. Based on the current studies, the neuron is the basic unit to dynamically process and propagate information [1]. Spiking Neural Network (SNN) is known as the third generation of neural network [2], which is enlightened by the behavior of neurons. Because of the bio-inspired philosophy of SNN, it has three distinguishable characteristics compared to artificial neural networks (ANNs). Firstly, the neuronal dynamics in SNN make it possible to simulate the real neuron system in the brain. Secondly, the additional temporal axis enables SNN to accept more inputs and build a more complex directed graph to propagate information. Thirdly, the event-driven based spike pattern makes SNNs widely deployed in neuromorphic devices for low-power brain-inspired computing.

Currently, SNNs are shown promising ability in processing dynamic and noisy information with high-efficiency [3, 4] and have been applied in a broad spectrum of tasks such as optical flow estimation [5], spike pattern recognition [6], SLAM [7], probabilistic inference [4], heuristically solving NP-hard problem [8], quickly solving optimization problem [9], sparse representation [10], robotics [11], and so forth.

1.1 Spiking Neural Network Training

How to train an SNN model with expected functionality is an essential problem for the SNN community. Many early studies have proposed unsupervised local learning based on the biological observation of local synaptic plasticity. In this family, spike timing-dependent plasticity (STDP) [12, 13, 14, 15, 16, 17, 18] has been widely explored, wherein each synaptic weight is modified locally based on the local spiking timing of the neurons wired by the synapse. However, such local synaptic plasticity suffers low accuracy and limited model scale, which is why its use in practical applications has been limited.

In order to improve the accuracy of SNNs, the algorithms in training ANNs are borrowed. Recently, an explicit format of gradient descent to train ANNs has been adapted and applied in training SNNs [19]. Due to the spatio-temporal data paths in SNNs, backpropagation through time (BPTT) is a good fit. Previous studies at the algorithm level have demonstrated the effectiveness of BPTT for SNN learning [20, 21, 22, 23, 6, 24], which can achieve higher accuracy.

Besides the functionality, how to train SNNs efficiently is also an important research topic. Currently, GPUs are still the mainstream platform for neural network training, while they are tailored for ANNs rather than SNNs. This can be reflected by the ANN-aware optimization for the GPUs' hardware architectures, programming libraries, training frameworks, etc. However, such optimization cannot fully utilize the special data format and computing paradigm of SNNs, thus causing inefficiencies when training SNNs on GPUs.

Beyond GPUs, researchers have also developed domain-specific chips for SNNs, usually termed as neuromorphic chips [25, 26, 27, 28, 29, 9, 30]. Here we focus on the ones targeting SNN learning rather than inference [31, 32, 33, 34, 35, 36]. Nearly all currently available SNN learning chips adopt local synaptic plasticity such as STDP for weight update. The good locality without backpropagation makes it easier to implement on decentralized many-core neuromorphic architectures. Although they enjoy low power and fast response, they still cannot escape from the low accuracy of these local learning rules. This is also one of the major reasons why neuromorphic chips have not yet achieved similar commercial success as deep learning accelerators.

1.2 Spiking Neural Network Security

With more attention to SNNs, the security problem becomes quite important. Here we focus on adversarial attack [37], one of the most popular threat models for neural network security. In adversarial attack, the attacker introduces imperceptible malicious perturbation into the input data to mislead the model's classification result. Although the adversarial attack is a well-studied topic in ANNs, it is still in its infant stage in the SNN domain.

For ANNs, the gradient-based attack method is the most efficient method [38, 39, 40, 41, 42, 43], however, there are several challenges in attacking an SNN model using the gradient-based methodology. First, the input gradient in SNNs presents as a spatio-temporal pattern that is hard to obtain with traditional learning algorithms like the gradient-free unsupervised learning [13, 18] and spatial-gradient-based ANN-to-SNN-conversion learning [44]. Second, the gradients are continuous values, incompatible with the binary spiking inputs. This data format incompatibility impedes the generation of spike-based adversarial examples via gradient accumulation. At last, there is severe gradient vanishing when the gradient crosses the step firing function with a zero-dominant derivative, which will interrupt the update of adversarial examples.

In the meantime, how to improve the robustness of an SNN under adversarial attack is important. CROWN-IBP [45, 46, 47] is one of the most promising certified training methods to improve the robustness of a neural network model. The CROWN-IBP method will compute the output boundary for a given bounded input. The core mission in CROWN-IBP certified training is to find the upper and lower bound function for each operation and find tight linear relaxation for non-linear operations. However, the current CROWN-IBP method cannot be directly applied to SNNs. Firstly, the neuron dynamic in SNNs is more complicated. Hence, new boundary functions should be defined to bound the unique non-linear operations in SNNs. Secondly, SNNs accept both spike and digital inputs, which requires additional boundary generalization for different input types.

1.3 Contributions

The goal of this dissertation is to design efficient BPTT-based training frameworks for SNNs and explore a robust SNN model against adversarial attack.

For BPTT-based SNN training, the complex neuron modeling and additional temporal axis make the training inefficient for GPUs. In order to improve training efficiency, this dissertation provides an accelerator solution and a GPU framework:

- Propose an accelerator (*H2Learn*) [48] which consists of a Forward Engine, a Backward Engine, and a Weight Update Engine. In Forward Engine and Weight Update Engine, a LUT-based design enables efficient multiplications and accumulations. In Backward Engine, an architecture that utilizes both the input and output sparsity is designed to reduce the computation overhead. See Chapter 3.
- Design a framework that can accelerate the BPTT-based training on GPUs [49]. The framework first optimizes the training dataflow to reduce the memory overhead. Then, kernel optimization is applied to the complex neuron modeling to reduce the kernel launching time. See Chapter 4.

In order to acquire a robust SNN model, this dissertation first explores the adversarial attack in SNNs. Then, the robust training with certified defense is designed to improve the SNN robustness.

- Design an algorithm to generate adversarial examples for SNNs efficiently [50]. The proposed algorithm can generate adversarial examples for both spike and digital inputs. Various techniques are designed to reduce the introduced perturbation on adversarial examples and handle the gradient vanishing problem during the attack. See Chapter 5.
- Introduce a robust training method for SNNs [51], which is enlightened by the certified defiense method. The proposed robust training method designs new boundary functions for each non-linear function in SNN. Also, the boundary for different input types is formalized. See Chapter 6.

Chapter 2

Background and Related Work

This chapter first provides a detailed elaboration of the BPTT-based SNN training. Then, several related studies on neuromorphic chips for both inference and training are introduced. A summary of the research on SNN security is presented at last.

2.1 Preliminary of Spiking Neural Networks

Compared to the early stage of unsupervised SNN training, BPTT-based learning algorithms exhibit extraordinary accuracy boost on the general tasks. This section presents a detailed algorithm description of the BPTT-based SNN training.

2.1.1 Neuron Modeling

In SNNs, a neuron is the basic structural unit as shown in Figure 2.1, which is comprised of dendrite, soma and axon. Many neurons connected by weighted synapses form an SNN, in which the binary spike events carry information for inter-neuron communication. Dendrite integrates the weighted pre-synaptic inputs, and soma consequently updates the membrane potential and determines whether to fire a spike or not. When

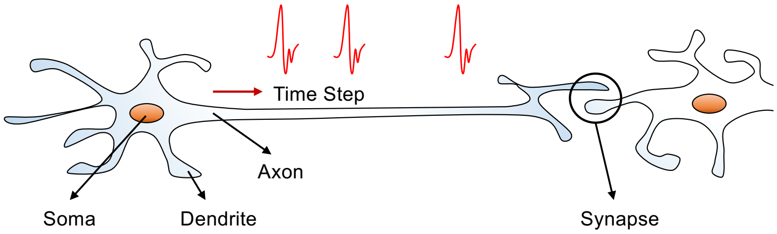


Figure 2.1: Neuronal Components

the membrane potential crosses a threshold, a spike will be fired and sent to post-neurons through axon.

In order to simulate the neuron behavior, this disertation adopts the well-studied leaky integrated-and-fire (LIF) for the neuron modeling [52]. Specifically, Figure 2.2(a) illustrates the behaviors of a spiking neuron. The input spikes are first weighted by synapses and then integrates by dendrites to update the state of membrane potential at soma. Once the membrane potential exceeds a threshold (th_f), the neuron fires a spike event to its post-connected neurons and resets its membrane potential to a reset value (usually zero); otherwise, nothing happens but the leakage of the membrane potential. A spiking neuron in SNNs is different from an artificial neuron in ANNs. Specifically, (1) there is an intrinsic temporal domain in a spiking neuron but not in an artificial neuron; (2) the membrane potential updating of a spiking neuron depends on both the historic state and the input integration, while the accumulated pre-activation in an artificial neuron just integrates inputs; (3) spiking neurons communicate with each other using binary spike events (0 or 1) while artificial neurons use continuous activations.

Fig. 2.2(b) shows a spiking network. The information propagates in both spatial and temporal domains. The output of an SNN is in a 2D spike pattern rather than a 1D vector in ANNs. The classification result is determined by the coding scheme of output. The rate coding [53, 54] is the commonly adopted one that the neuron fires the most indicates the recognized class. Notice that the network structure of SNNs can be arbitrary

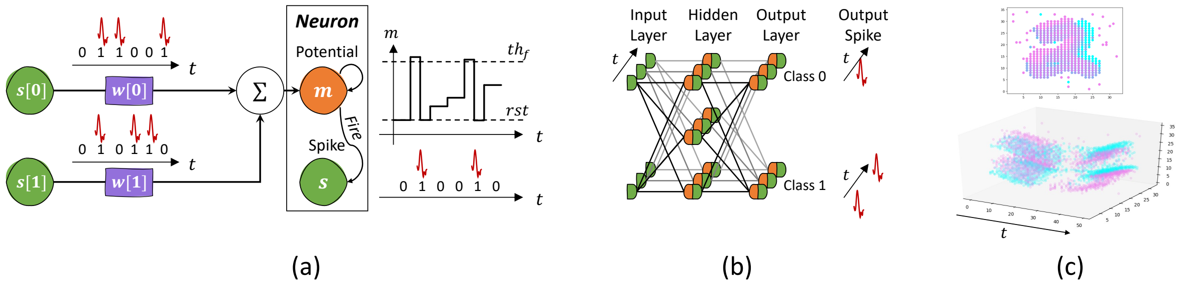


Figure 2.2: Introduction of SNNs: (a) behavior of a single spiking neuron; (b) a spiking neural netowrk; (c) the input format.

in principle, but the fully-connected (FC) layers based multilayered perceptron (MLP) and the convolutional (Conv) layers based convolutional neural network (CNN) are two usual cases, which is similar to ANNs. Benefited from the binary format of neuronal inputs/outputs, the multiplier-based matrix operations can be eliminated during SNN inference, which becomes the source of high efficiency in SNN hardware design.

2.1.2 BPTT-based Spiking Neural Network Training

Since BPTT-based SNN training can improve the accuracy of SNNs dramatically, this dissertation focuses on the BPTT-based training algorithms. There are three stages during the training: forward pass (FP), backward pass (BP), and parameter update (PU). The FP stage is illustrated in Figure 2.3(a). Specifically, the LIF dynamics is governed by

$$\boldsymbol{s}_t^l = fire(\boldsymbol{m}_t^l - th_f), \tag{2.1}$$

$$\boldsymbol{m}_t^l = \underbrace{\alpha \cdot \boldsymbol{m}_{t-1}^l \cdot (1 - \boldsymbol{s}_{t-1}^l)}_{temporal} + \underbrace{\boldsymbol{x}_t^l}_{spatial}, \tag{2.2}$$

Here, $\boldsymbol{m}_t^l$ and $\boldsymbol{s}_t^l$ represent the membrane potentials and the spike events of the neurons in the l-th layer at the t-th time step. The neuron would fire a spike and send it to the post-synaptic neurons once the membrane potential larger than the firing threshold th_f. $fire(\cdot)$ is the Heaviside step function, i.e., $fire(x) = 1$ if $x \geq 0$; $fire(x) = 0$ otherwise. The membrane potential is comprised by the temporal part and the spatial part. In the temporal part, if the neuron did not fire a spike in the previous time step, the membrane potential decays by a factor α, otherwise, the it will be reset to 0. The spatial part $\boldsymbol{x}$ is the result after CONV/FC which follows

$$\boldsymbol{x}_t^l[j] = \sum\nolimits_k \boldsymbol{s}_t^{l-1}[k] * \boldsymbol{w}^l[k,j] + \boldsymbol{b}^l[j]. \tag{2.3}$$

Conv/FC takes the spike events of the previous layer as inputs, wherein the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ are trainable parameters.

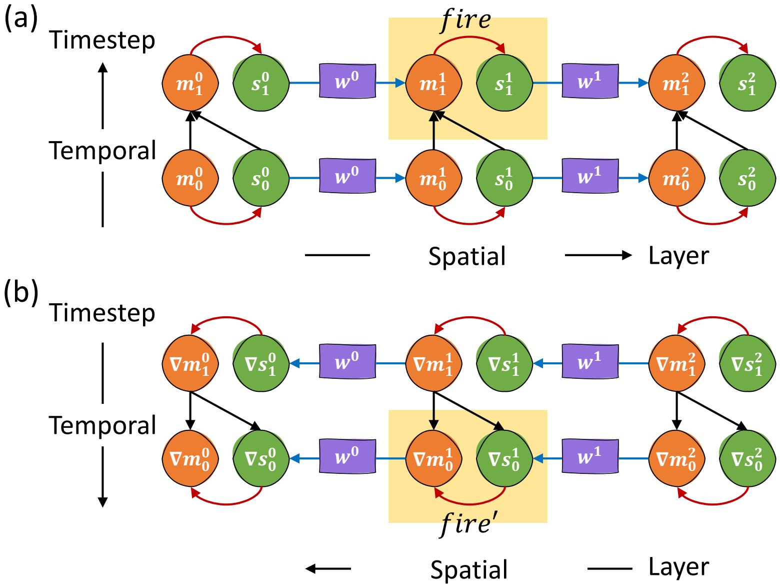

(a)
Timestep
Temporal
fire
m_1^0 s_1^0 w^0 m_1^1 s_1^1 w^1 m_1^2 s_1^2
m_0^0 s_0^0 w^0 m_0^1 s_0^1 w^1 m_0^2 s_0^2
Spatial
Layer
(b)
Timestep
Temporal
∇m_1^0 ∇s_1^0 w^0 ∇m_1^1 ∇s_1^1 w^1 ∇m_1^2 ∇s_1^2
∇m_0^0 ∇s_0^0 w^0 ∇m_0^1 ∇s_0^1 w^1 ∇m_0^2 ∇s_0^2
fire'
Spatial
Layer

Figure 2.3: Information propagation path of (a) the forward pass and (b) the backward pass in BPTT for SNN learning.

In the BP stage, the gradients propagate along the opposite direction of the arrows in Figure 2.3(a), which can be illustrated with Figure 2.3(b). The gradients of the spikes and the membrane potentials are calculated as

$$\nabla \boldsymbol{s}_t^l[k] = \underbrace{\nabla \boldsymbol{m}_{t+1}^l[k] \cdot (-\alpha \cdot \boldsymbol{m}_t^l[k])}_{temporal} + \underbrace{\textstyle\sum_j \nabla \boldsymbol{x}_t^{l+1}[j] * \boldsymbol{w}^{l+1}[k,j]}_{spatial}, \tag{2.4}$$

$$\nabla \boldsymbol{m}_t^l = \nabla \boldsymbol{m}_{t+1}^l \cdot \alpha \cdot (1 - \boldsymbol{s}_t^l) + \nabla \boldsymbol{s}_t^l \cdot fire'(\boldsymbol{m}_t^l). \tag{2.5}$$

The spike gradients $\nabla \boldsymbol{s}_t^l$ are obtained from both temporal and spatial directions. The temporal part is derived from the temporal part of Equation 2.2, while the spatial part is the gradient format of Conv/FC in Equation 2.3, where $\nabla \boldsymbol{x} = \nabla \boldsymbol{m}$. The membrane potential gradients $\nabla \boldsymbol{m}_t^l$ are acquired by calculating the partial derivative of Equation 2.1-2.2. The derivative of the $fire$ function does not exist in principle. We adopt an approximation [20] to simulate the derivative:

$$fire'(\boldsymbol{m}_t^l[i]) \approx \begin{cases} \eta, & th_l < \boldsymbol{m}_t^l[i] < th_r, \\ 0, & otherwise, \end{cases} \tag{2.6}$$

where η is a decay factor.

In the PU stage, the parameter update is derived from Equation 2.3, which follows

$$\nabla \boldsymbol{w}^l[k,j] = \sum\nolimits_t \nabla \boldsymbol{x}_t^l[j] * \boldsymbol{s}_t^{l-1}[k], \tag{2.7}$$

$$\nabla \boldsymbol{b}^l[j] = \sum\nolimits_{t,n_j} \nabla \boldsymbol{x}_t^l[n_j]. \tag{2.8}$$

Finally, the SNN model will be updated with the computed parameter gradients.

Compared with the traditional unsupervised learning rules, such as STDP, BPTT-based SNN learning algorithms can always achieve higher accuracy and scalability. An accuracy comparison between these two learning algorithm is shown in Table 2.1.2.

Method	Ref	Dataset	Net	Accuracy
STDP	[14]	MNIST	CNN	93.30%
	[13]	MNIST	MLP	95.00%
	[16]	MNIST	CNN	97.50%
	[15]	MNIST	CNN	98.40%
	[55]	CIFAR10	CNN	63.64%
	[56]	CIFAR10	CNN	75.20%
	[57]	CIFAR10	CNN	66.23%
BPTT	[24]	MNIST	MLP	98.60%
	[20]	MNIST	MLP	98.89%
	[21]	MNIST	CNN	99.49%
	[21]	N-MNIST	MLP	98.88%
	[23]	N-MNIST	CNN	99.20%
	[6]	N-MNIST	CNN	99.44%
	[6]	CIFAR10	CNN	89.83%
	[6]	CIFAR10-DVS	CNN	58.10%

Table 2.1: Accuracy comparison for SNN learning: STDP vs. BPTT.

2.2 Related work on SNN Acceleration

2.2.1 Chips for SNN Inference

Many neuromorphic chips target SNN inference. Some of them adopt mixed-analog-digital circuits based designs [31, 33, 58] that are usually power efficient but suffer low accuracy and poor programmability. The modern neuromorphic chips prefer fully digital designs [32, 34, 35, 36, 59]. In particular, TrueNorth [32] achieves low power via event-driven asynchronous circuits; Tianjic [34, 35] bridges ANNs and SNNs using a hybrid architecture with a unified routing infrastructure; Spinalflow [36] designs an accelerator that can skip redundant computations via input scattering. Different from them for SNN inference, *H2Learn* targets SNN learning.

2.2.2 Chips for SNN Training

Most of SNN learning chips are designed to implement local synaptic plasticity rules. Similarly, there are also analog circuits based designs [25, 27] and digital solutions [28, 29, 9, 30]. Specifically, ODIN [28] is the digital version of ROLLS [27] with only one core per chip, and MorphIC [29] is an enhanced version with a hierarchical routing topology; Loihi [9] adopts a many-core architecture, while FlexLearn [30] further extends the scope of synaptic plasticity rules. Some studies exploit either SNN inference or training on FPGA [60, 61, 62]. Unlike implementing the local synaptic plasticity rules with lower accuracy, *H2Learn* selects the BPTT learning rule to achieve high accuracy and elaborates the architecture to achieve high efficiency. We also notice a recent work [63] supporting BP (not BPTT) for SNNs, but it adopts a LIF variant without temporal propagation, focuses on exploiting the non-volatile memory technology, and only shows results on the small MNIST dataset with two FC layers.

2.2.3 Accelerating on GPUs

Also, recent studies explore accelerating SNN training on GPUs [64]. However, current GPU platform is hard to utilize the binary character of spike and high sparsity during SNN training. Besides learning chips for SNN, many training accelerators target ANNs [65, 66, 67, 68]. Because SNNs involve distinct operand characteristics, e.g., the more complex neuronal dynamics and the additional timestep dimension, those ANN-oriented accelerators cannot be directly applied for SNN training.

2.3 Related work on SNN Security

Recently, researchers attempt to evaluate the well-studied attack methods in ANNs on SNNs [69], such as PGD, SparseFool, and adversarial patch. Instead of attacking SNN directly, a model conversion method can transfer attack an SNN to the counterpart ANN [70]. Besides gradient-based attack methods, some gradient-free attack methods are designed that only need to know the input, such as trial-and-error perturbation [71] and dash attack [72]. However, these attack methods either suffer high computational complexity or do not compatible with various input formats.

Researchers also investigated the impact of hyper-parameter selection [73] and input filtering [72] on the adversarial attack in SNNs. However, these methods do not directly promote the classification behavior of a given SNN model.

Chapter 3

H2Learn: High-Efficiency Learning Accelerator for High-Accuracy Spiking Neural Networks

Although BPTT based SNN training achieves much higher accuracy than traditional unsupervised learning algorithms, current general-purpose processors suffer from low training efficiency when performing BPTT for SNNs due to the ANN-tailored optimization.

This Chapter introduces *H2Learn*, a novel architecture that can achieve high efficiency for BPTT-based SNN learning which ensures high accuracy of SNNs. In the beginning, we characterized the behaviors of BPTT-based SNN learning. Benefiting from the binary spike-based computation in the forward pass and the weight update, we first design look up table (LUT) based processing elements in Forward Engine and Weight Update Engine to make accumulations implicit and to fuse the computations of multiple input points. Second, benefited from the rich sparsity in the backward pass, we design a dual-sparsity-aware Backward Engine which exploits both input and output sparsity. Finally, we apply a pipeline optimization between different engines to build an end-to-end solution for the

BPTT-based SNN learning. Compared with the modern NVIDIA V100 GPU, *H2Learn* achieves 7.38× area saving, 5.74-10.20× speedup, and 5.25-7.12× energy saving on several benchmark datasets.

3.1 Overview and Motivation

Currently, two bottlenecks hinder the progress of SNNs: (1) low accuracy of conventional local learning rules (e.g., STDP), limiting their competitiveness and application scope in practice; (2) low execution efficiency on GPUs, limiting the exploration of the model scale and space (indirectly limiting accuracy). The former can be significantly improved by the BPTT algorithm, and the latter is due to GPUs' specific optimization for ANNs, rather than for SNNs with special data format and computing paradigm. Therefore, we propose to design an efficient accelerator for BPTT-based SNN training to improve the competitiveness of neuromorphic chips.

3.1.1 Low-Accuracy SNN Learning on Neuromorphic Chips

In order to build high-efficiency domain-specific chips for SNN learning, researchers have designed neuromorphic chips. However, almost all of them [25, 26, 27, 28, 29, 9, 30] adopt unsupervised learning rules inspired by bio-plausible synaptic plasticity, such as STDP [12], the good locality of which makes it hardware friendly. However, in practical applications, this rule cannot be accepted due to the low accuracy when performing mainstream tasks (see Table 2.1.2) and the difficulty in scale-up when encounter complex tasks, which are the major reasons why neuromorphic chips are suffering skepticisms and are not applied widely as deep learning accelerators.

3.1.2 Low-Efficiency SNN Learning on GPUs

GPUs play the backbone role in neural network training. However, current GPU hardware architectures (e.g., tensor cores on NVIDIA GPUs), programming libraries (e.g., cuDNN), and model training frameworks (e.g., TensorFlow and Pytorch) are mainly optimized for ANNs rather than SNNs, causing inefficiencies when training SNN models.

Specifically, there are mainly three inefficiencies on GPUs to train an SNN model with the BPTT learning algorithm. First, GPUs can only accelerate either high-precision computation (e.g., FP16, FP32) or low-precision computation (e.g., INT1, INT8, INT16); however, the major operations such as Conv and FC in SNN training involve both binary operands and floating-point operands, which is not optimized by GPUs. Second, the sparsity optimization libraries on GPUs are hard to bring real acceleration if the sparsity pattern is irregular. For example, according to the official white paper of cuSPARSE [74], GPUs can achieve speedup only when the irregular sparsity is greater than 95%. The last inefficiency is the lack of SNN-training-oriented dataflow optimization on GPUs. In SNNs, the specific neuron model (with temporal dynamics and bivariate iteration) and an additional timestep dimension are involved, such that a more complex dataflow distinct from that of ANNs should be considered to reduce the memory footprint.

Table 3.1.2 shows that SNN training is much slower than ANN training on GPU under the same network structure.

Dataset	**Latency of ANNs**	**Latency of SNNs**	**Performance Drop**
MNIST	12.12s	138.55s	11.43×
CIFAR10	12.98s	147.07s	11.33×
ImageNet	0.72hr	7.47hr	10.37×

Table 3.1: Latency of one training epoch for ANNs and SNNs under the same network structure on NIVIDIA V100 GPU.

3.1.3 High-Accuracy and High-Efficiency SNN Learning

BPTT Learning for High Accuracy. Recently, researchers began to borrow ideas from the learning of ANNs. Typically, the gradient-descent-based backpropagation algorithms have been applied in SNN training [19, 20, 21, 22, 23, 6, 24], among which the backprogation through time (BPTT) algorithm has become an effective way to train SNN models with high accuracy via global optimization. Table 2.1.2 lists some reported accuracies for SNNs learnt by STDP and BPTT. Apparently, BPTT shows superior accuracy. For the more difficult datasets like CIFAR10, ImageNet, and CIFA10-DVS, STDP cannot provide good results while BPTT can do. Recent studies [75, 76] also try to reveal the connection between backpropagation and the brain, which is interesting but out of the scope of this dissertation.

Spike-based and Sparse Computing for High Efficiency. BPTT is a costly learning algorithm with backpropagation across the entire network and all timesteps. Fortunately, we find opportunities after a detailed algorithm profiling.

Dataset	**Layer**	**conv1**	**conv2**	**conv3**	**conv4**	**conv5**	**conv6**
MNIST	O sparsity (%)	22.66	85.99	52.74	74.74	53.82	-
Acc: 99.67%	I sparsity (%)	99.18	97.46	98.76	97.44	97.93	-
CIFAR10	O sparsity (%)	54.06	83.22	80.22	82.75	87.24	-
Acc: 87.63%	I sparsity (%)	91.41	83.89	89.92	88.63	84.39	-
N-MNIST	O sparsity (%)	59.89	86.13	92.95	-	-	-
Acc: 98.97%	I sparsity (%)	97.24	97.36	99.26	-	-	-
CIFAR10-DVS	O sparsity (%)	62.93	88.23	82.07	-	-	-
Acc: 63.00%	I sparsity (%)	87.69	74.14	95.77	-	-	-
ImageNet	O sparsity (%)	17.97	16.21	14.65	19.82	16.65	15.09
Acc: 60.90%	I sparsity (%)	94.95	94.11	92.94	96.02	94.38	93.35

Table 3.2: Sparsity in the backward pass of BPTT during SNN learning. Abbreviation: I-input, O-output.

There are three phases in the BPTT learning: forward pass, backward pass, and weight update. In the forward pass and weight update, one of the two operands for the computational operations is a binary spike, which implies that the costly multiplication units are no longer needed. In the backward pass, although multiplication operations cannot be avoided, there is rich sparsity that can be exploited. On one side, one of the operands for the computational operations in the backward pass is the membrane potential gradient that is sparse; on the other side, the outputs are the gradients of spike activities, a part of which will be zeroed out when backpropagating through the firing function. The detailed BPTT for SNN learning can be found in Section 2.1.2. In Table 3.1.3, we present the input and output sparsity values of each layer during backpropagation. It can be seen that the sparsity is quite rich, indicating opportunities to reduce compute and storage.

3.1.4 Analysis of BPTT for SNN Learning

A detailed BPTT-based learning algorithm is presented in Section 2.1.2. Based on the training process, we have the following observations:

- The spatial parts in Equation (2.2) & (2.4) and the weight gradient calculation in Equation (2.7) require the Conv or matrix multiplication (MM) operation in a Conv or FC layer, respectively. Other operations are element-wise, which have much fewer workloads. Therefore, the architecture design of *H2Learn* makes more efforts to accelerate the costly Conv or MM operations in the context of SNN learning.

- The spikes are in the binary format, i.e. either 0 or 1. It is efficient to store the spike data in a compact format and use LUT-based operation to reduce computation.

- Based on the $fire'(\cdot)$ in Equation (2.6), we can determine the valid neurons (marked by $\nabla\tilde{s}$) that allow the gradient to pass through during the backward pass, according to their membrane potential values in the forward pass. Specifically, when a neuron's membrane potential is within $[th_l, th_r]$ in the forward pass, it is valid and needs to calculate its spike gradient in the backward pass; otherwise, we can skip the computation (termed as output sparsity in the Backward Engine design, see Table 3.1.3). During forward pass, there is also no need to store the membrane potentials (for the use in the temporal part of Equation (2.6)) of invalid neurons.

- The goal of learning is to update weights of the model via calculating weight gradients. From Equation (2.7), we find that $\nabla \boldsymbol{w}$ only requires membrane potential gradients and does not involve spike gradients. Therefore, $\nabla \boldsymbol{s}$ can be treated as intermediate data and merged into the ∇u calculation.

Stage	**Inputs**	**Outputs**	Equations	Major Operation
FP	m_{t-1}^l, s_{t-1}^l, s_t^{l-1}, w^{l-1}	s_t^l , m_t^l , $\nabla\tilde{s}_t^l$	Equation (2.3)	$\sum_j s_t^{l-1}[j]w^{l-1}[j,i]$, spike-based OP
WU	∇m_t^{l+1} , s_t^l	∇w^l	Equation (2.7)	$\sum_t \nabla m_t^{l+1}[j]s_t^l[i]$, spike-based OP
BP	∇m_{t+1}^l , m_t^l, ∇m_t^{l+1}, $\nabla\tilde{m}_t^{l+1}$, w^l, $\nabla\tilde{s}_t^l$, s_t^l	∇m_t^l, $\nabla\tilde{m}_t^l$	Equation (2.4)	$\sum_j \nabla m_t^{l+1}[j]w^l[i,j]$, sparse FP16 OP

Table 3.3: I/O and the major operation of an SNN layer for each learning stage. The variables in the binary format are marked in red.

The inputs, outputs, and the major operation of an SNN layer for each training stage are listed in Table 3.3. The variables in the binary format are marked in red. We use bit masks $\nabla\tilde{\boldsymbol{s}}$ and $\nabla\tilde{\boldsymbol{m}}$ to indicate which neurons have valid spike gradients and non-zero membrane potential gradients in the backward pass, respectively.

3.2 Architecture of *H2Learn*

In this section, we first introduce how to handle data with different formats and then detail the architecture design for each engine in *H2Learn*. Unlike the training accelerators for ANNs that usually adopt one engine for all training stages [77, 78], we design different engines for each SNN training stage. The philosophy behind this design is that the behavior of each training stage is distinct from each other. Specifically, (1) different data formats, i.e., one of the operands in the forward pass and weight update is a spike, however, all operands in the backward pass are real values; (2) different computation characteristics, i.e., the forward pass and weight update can take benefits from spikes to improve efficiency, however, the backward pass utilizes the input and output sparsity to simplify computation; (3) different dataflows, i.e., the Conv (or MM) dataflows in the forward and backward passes are distinct from that in weight update. Based on these special features, that are distinct from ANNs, we design a Forward Engine, a Weight Update Engine, and a Backward Engine, which form the backbone of our *H2Learn*. Finally, these engines can be pipelined during training to gain optimized overall performance.

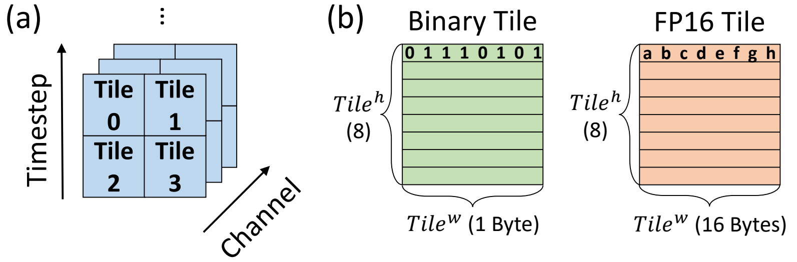


Figure 3.1: Illustration of feature map (FM) tiling: (a) an example of FM tiling; (b) configuration of a tile with different data types.

Figure 3.1 shows the feature map (FM) tiling. The dimension of the FMs is $T \times C \times H \times W$, where T represents the total number of timesteps, C, H and W stand for the channel, height, and width, respectively. For an FM locating at timestep t and channel c, we split it into several tiles as shown in Figure 3.1(a), and the tile corresponds to the basic handling data unit in our design. In this work, we need to consider two data types: binary spike data and floating-point 16-bit (FP16) data. Figure 3.1(b) shows the example of a tile in the two data formats.

3.2.1 Forward Engine

We design Forward Engine to handle the forward pass in BPTT learning. From Table 3.3, the Conv of the spatial part in Equation (2.2) is the most costly operation, which is shown in Figure 3.2(a). Each time, Forward Engine takes $T_{max}^{PE} \times C_{s^{l-1}}^{PE}$ tiles from s_t^{l-1} and performs Conv with a part of weights whose size is $k^2 \times C_{s^{l-1}}^{PE} \times C_{m^l}^{PE}$ to produce $T_{max}^{PE} \times C_{m^l}^{PE}$ tiles of partial sums ps_t^l which belong to the spatial part of u_t^l. $C_{s^{l-1}}$ and C_{m^l} are the numbers of channels of s^{l-1} and m^l, respectively. T_{max}^{PE}, $C_{s^{l-1}}^{PE}$, and $C_{m^l}^{PE}$ represent the maximum number of timesteps, channels of s^{l-1} and m^l that Forward Engine can process at one time. k denotes the weight kernel size. In this work, we call such processing as one grid iteration.

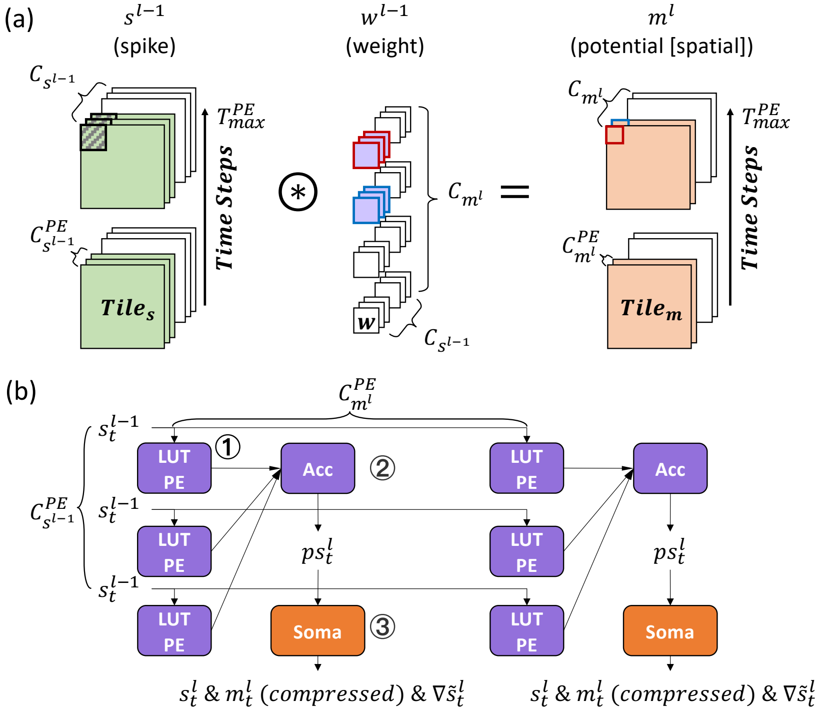


Figure 3.2: Forward Engine: (a) operation; (b) microarchitecture.

Figure 3.2(b) shows the microarchitecture of Forward Engine. In this example, the layout of the PE array is $C_{s^{l-1}}^{PE}$ rows by $C_{m^l}^{PE}$ columns. In Forward Engine, each row shares the same $Tile_s$ from s^{l-1} and each column contributes to the same $Tile_m$ in m^l. The workflow of Forward Engine in a Conv layer includes following steps: ① the PE array receives sliding windows from s_t^{l-1} and performs the LUT-based Conv (detailed later); ② each accumulator (Acc) integrates outputs from PEs of the same column; ③ when the partial sum ps_t^l includes all $C_{s^{l-1}}$ channels, the result will be sent to Soma to get s_t^l, m_t^l (abandoned if out of $[th_l, th_r]$), and $\nabla \tilde{s}_t^l$ (needed in the backward pass). The processing of different sliding windows, timesteps, and samples reuses the PE array resource.

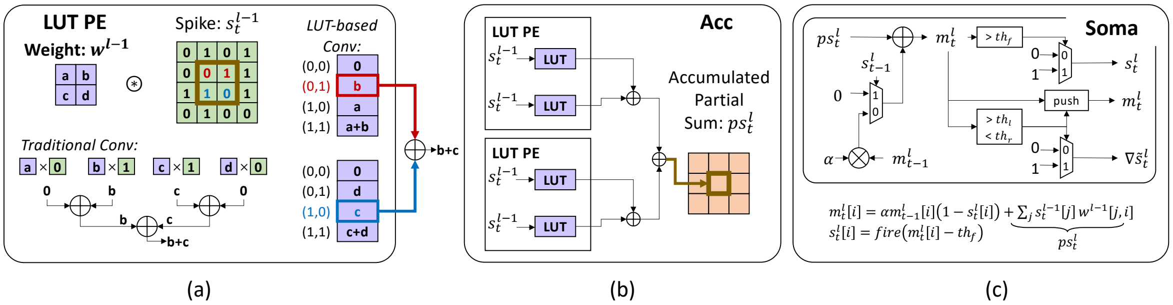


Figure 3.3: Details of each function unit in Forward Engine: (a) LUT PE that performs a part of LUT-based Conv; (b) Acc unit that performs the rest of LUT-based Conv via inter-PE accumulation; (c) Soma unit that produces the spike, compressed membrane potential , and spike gradient mask.

Figure 3.3 details each block. Figure 3.3(a) shows how to realize spike Conv using LUT. In this example, we perform a Conv between a 2 × 2 weight kernel and a sliding window in s_t^{l-1}. The 2D Conv is traditionally executed as a dot product. However, the inputs are binary spikes in SNNs, thus each sliding window can be represented as one of fixed states, i.e., 2^n states for n elements. Here, 4 binary input elements in a sliding window have 16 patterns. We can calculate the Conv results for all possible patterns in advance and store them in an LUT. With this design, we use the input spikes as an access address to load results from the LUT.

However, the LUT-based solution might increase the data need to store. We mitigate the storage consumption by splitting a large LUT into several small sub-LUTs. In Figure 3.3(a), we use two sub-LUTs to cover different regions of the sliding window, and the LUT size in one PE can be reduced from 16 to 8. Notice that one extra adder is required to accumulate the partial results from sub-LUTs. We call this kind of PE units as LUT PE. Section 3.3.2 shows the detailed analyses for optimal LUT setting. In our design, each LUT PE loads the partial Conv results from all of its sub-LUTs, and the Acc unit accumulates the outputs from the LUT PEs in the same column using an adder tree with FP16 precision as Figure 3.3(b).

After finishing the spatial part compute, we feed the final ps_t^l to the Soma unit which will update the membrane potential and determine whether to fire a spike s_t^l or not as in Figure 3.3(c). According to our previous analysis, during the backward pass, we only need to store the membrane potentials of valid neurons whose membrane potentials fall into $[th_l, th_r]$. Therefore, Soma generates a compressed u_t^l without storing zero elements and also produces the corresponding binary spike gradient mask $\nabla \tilde{s}_t^l$:

$$\nabla \tilde{s}_t^l[i] = \begin{cases} 1, & th_l < m_t^l[i] < th_r, \\ 0, & otherwise. \end{cases} \tag{3.1}$$

For FC layers, the weight matrix size is $C_{s^{l-1}} \times C_{m^l}$. Each column of the PE array belongs to a m^l channel. We can treat the sub-LUTs in the same column as weight buffers that contain weights of the same output channel but many input channels. The number of input channels for each PE row relies on the size of sub-LUTs in each PE. During processing, each sub-LUT exports a weight element to Acc if the corresponding input spike is 1; otherwise exports 0. We do not compress m_t^l in FC layers during the forward pass, since the data volume is far smaller than that in Conv layers. The pooling layer is integrated into the Soma unit.

3.2.2 Weight Update Engine

We design Weight Update Engine to calculate the weight gradient. From Equation 2.7, the weight gradient is calculated by performing a Conv between s_t^l and ∇m_t^{l+1}, as shown in Figure 3.4(a). Weight Update Engine takes $T_{max}^{PE} \times C_{s^l}$ $Tile_s$ as inputs and performs Conv with $T_{max}^{PE} \times C_{\nabla m^{l+1}}^{PE}$ $Tile_m$ to produce the partial sum ps_t^l of ∇w^l whose size is $k^2 \times C_{s^l} \times C_{\nabla m^{l+1}}^{PE}$.

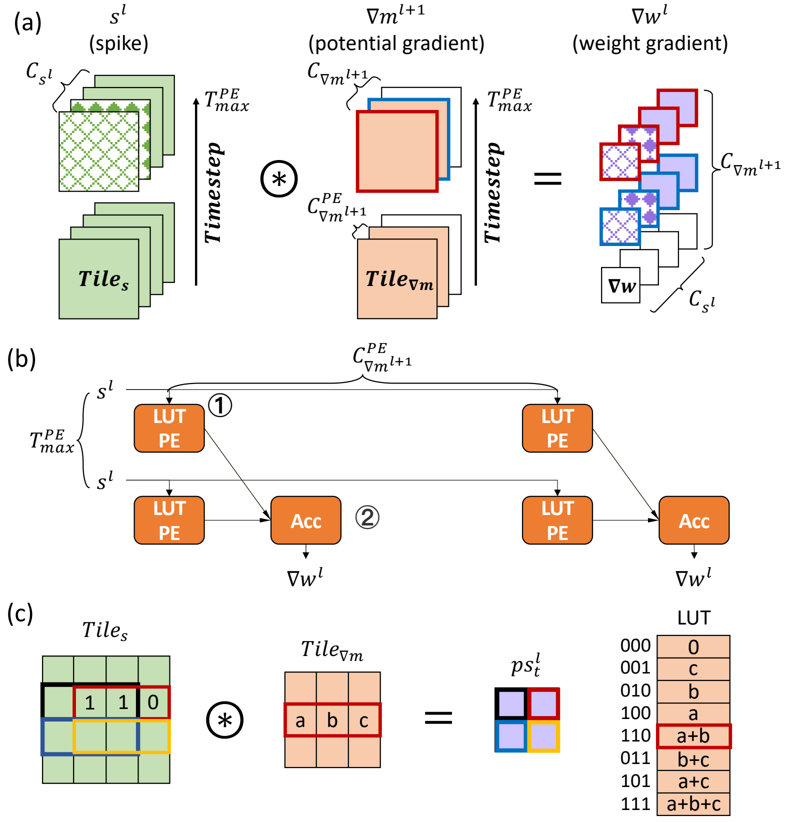


Figure 3.4: Weight Update Engine: (a) operation; (b) microarchitecture; (c) LUT-based Conv.

Figure 3.4(b) presents the microarchitecture of Weight Update Engine. The numbers of rows and columns correspond to T_{max}^{PE} and $C_{\nabla m^{l+1}}^{PE}$, respectively. Such correspondence is different from the Forward Engine, since the column dimension is the input FM channel in the Forward Engine while is the timestep in the Weight Update Engine. The workflow of Weight Update Engine includes two steps: ① performs LUT-based Conv between s^l and ∇m^{l+1} in the PE array; ② accumulates the outputs from LUT PEs of the same column in the Acc unit. Notice that the processing of different sliding windows and input channels reuses the PE array resource. Since s^l is in the binary format, we can still use the LUT-based Conv as in the Forward Engine. Figure 3.4(c) shows an example of LUT-based Conv in the Weight Update Engine.

For FC layers, the weight gradient calculation requires a dot product between two matrices whose sizes are $C_{s^l} \times T_{max}^{PE}$ and $T_{max}^{PE} \times C_{\nabla m^{l+1}}$. We directly feed elements in ∇m^{l+1} to the sub-LUTs. Every PE row shares inputs at the same timestep. Similar to Forward Engine, each sub-LUT exports an element to the Acc unit based on the state of the spike input.

3.2.3 Backward Engine

We design Backward Engine to compute the membrane potential gradient in the backward pass. From table 3.3, Conv is the major operation, as shown in Figure 3.5(a). Backward Engine takes $T_{max}^{PE} \times C_{\nabla m^{l+1}}^{PE}$ $Tile_{\nabla m}$ and performs Conv with corresponding weight kernels (after 180 degree rotation) whose size is $k^2 \times C_{\nabla m^{l+1}}^{PE} \times C_{s^l}^{PE}$ to generate $T_{max}^{PE} \times C_{s^l}^{PE}$ tiles of partial sum ps_t^l for ∇s_t^l. The operand data type here is FP16, which increases the compute cost. However, we can exploit the output sparsity (indicated by $\nabla \tilde{s}_t^l$ pre-generated in the Soma unit during the forward pass). Moreover, we can utilize the input sparsity in ∇m to further simplify computation. Specifically, we use a bitmap $\nabla \tilde{m}$ to record the input sparsity information:

$$\nabla \tilde{m}_t^l[i] = \begin{cases} 1, & \nabla m_t^l[i] \neq 0, \\ 0, & otherwise. \end{cases} \tag{3.2}$$

From the hardware perspective, we use input and output neuron IDs in a tile to locate valid operands according to the corresponding bitmaps, which will be introduced latter.

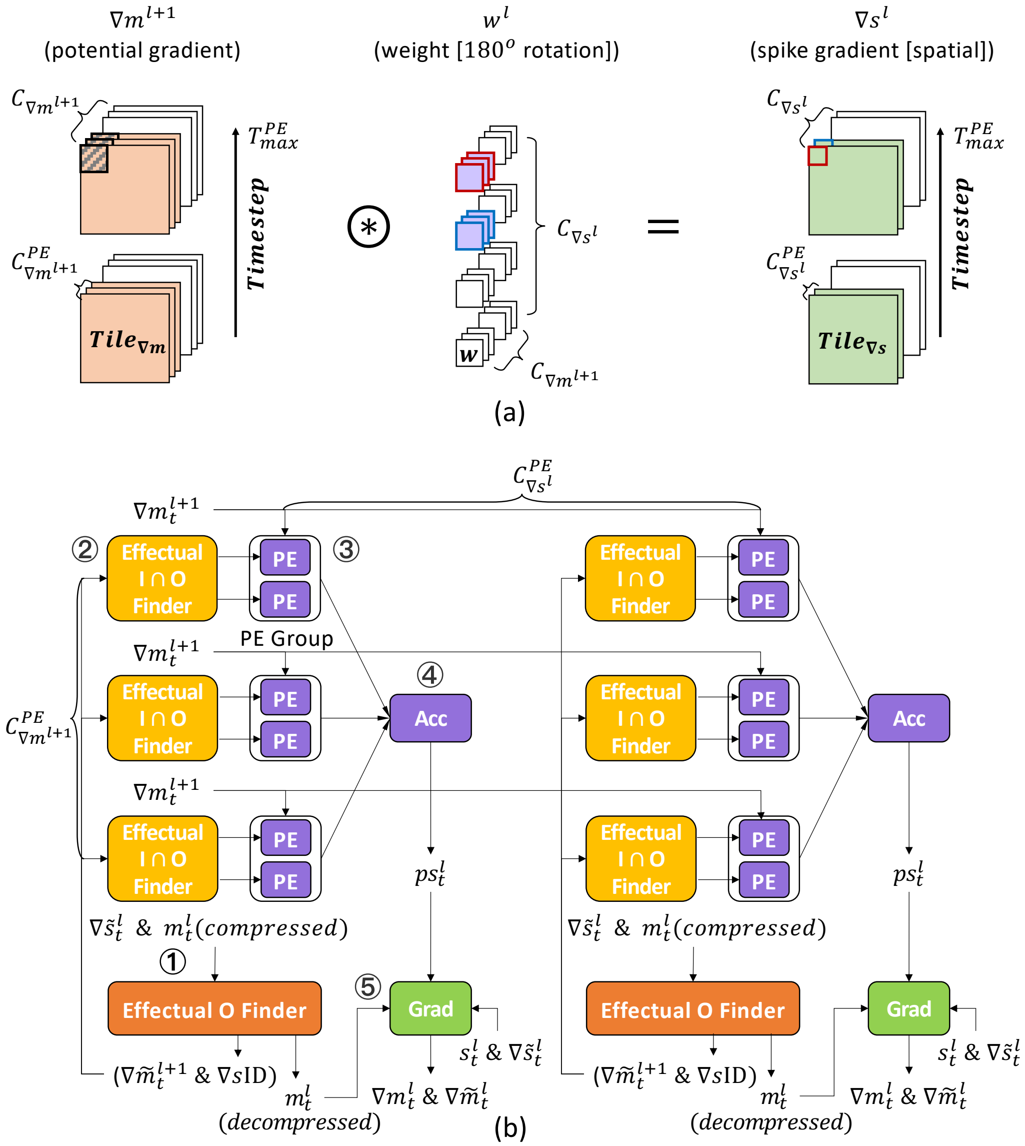


Figure 3.5: Backward Engine: (a) operation; (b) microarchitecture.

The microarchitecture of Backward Engine is shown in Figure 3.5(b). The PE array layout is $C^{PE}_{\nabla m^{l+1}}$ rows by $C^{PE}_{s^l}$ columns. PEs in the same row share the same $Tile_{\nabla m}$ from m^{l+1} and the corresponding $\nabla \tilde{m}_t^{l+1}$. PEs in the same column generate the partial sum ps_t^l of the spatial part of the same $Tile_{\nabla s}$ of s^l, also these PEs share the same $\nabla \tilde{s}_t^l$.

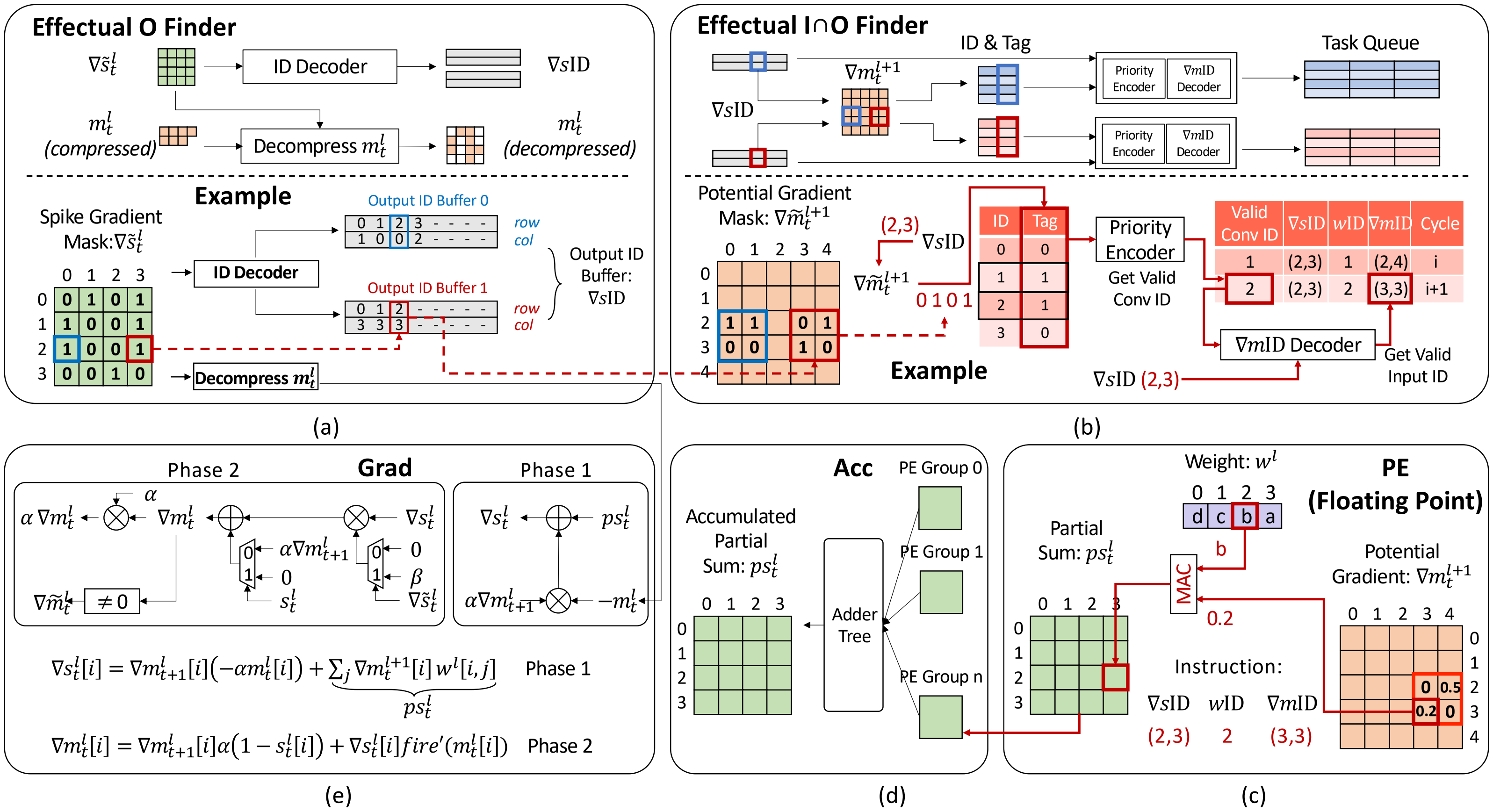


Figure 3.6: Details of each function unit in Backward Engine: (a) Effectual Output Finder unit finds neuron IDs that have valid ∇s and decompresses u_t^l, exploiting output sparsity; (b) Effectual Input∩Output Finder unit generates valid microinstructions for each Conv operation in the PE unit, exploiting both input and output sparsity; (c) & (d) PE and Acc units that perform the valid FP16 MACs in the Conv operation; (e) Grad unit that produces ∇u_t^l and $\nabla \tilde{u}_t^l$.

The workflow of Backward Engine in a Conv layer has the following steps: ① Effectual O Finder gets the valid output neuron IDs (∇sIDs) for Conv according to $\nabla \tilde{s}_t^l$, and decompresses the compressed m_t^l into the original dense format; ② Effectual I∩O Finder further generates valid input and output neuron IDs for Conv according to $\nabla \tilde{m}_t^{l+1}$ and the above ∇sIDs; ③ PEs perform Conv that exploits both input and output sparsity; ④ the Acc unit accumulates partial sums from PEs of the same column; ⑤ the Grad unit calculates ∇s_t^l and ∇m_t^l. Finally, the produced ∇m_t^l and the corresponding $\nabla \tilde{m}_t^l$ serve as outputs. The processing of different sliding windows, timesteps (T_{max}^{PE}), and samples reuses the PE array resource.

Figure 3.6 shows each function unit in Backward Engine. Since the amount of workloads of Conv in membrane potential gradient calculation is far larger than that in other stages, we consider multi-PEs (four-PEs in our experiments) in a PE group to improve the parallelism of processing a tile. The first functionality of the Effectual O Finder unit is to get the valid output neuron IDs (∇sIDs) according to the stored spike gradient mask $\nabla \tilde{s}_t^l$ in the forward pass. As an example shwon in Figure 3.6(a), we take two Output ID Buffers to store the coordinates of ∇sIDs, which are shared by the PE groups in the same column. ID Decoder scans the elements in $\nabla \tilde{s}_t^l$ row by row, and then writes ∇sIDs into Output ID Buffers alternatively. In this way, the two buffers can save close amount of ∇sIDs that need to be processed by PEs in a PE group. Each PE in a PE group would process the workloads stored in one of the buffers, thus mitigating the workload imbalance between PEs in a PE group. The second functionality of Effectual O Finder is to decompress m_t^l that is stored in a compressed form during the forward pass. This decompression can make the element-wise operations in the Grad unit easier to execute.

Then, each Effectual O Finder unit sends ∇sIDs to the Effectual I∩O Finder units in the same column to search valid multiplications in Conv (termed as valid Conv IDs) wherein both input ($\nabla \tilde{m}_t^{l+1}$) and output ($\nabla \tilde{s}_t^l$) are valid (i.e., non-zero). The procedure is shown in Figure 3.6(b). We process two Output ID Buffers separately (marked in blue and red). For example, computing the point ∇sID=(2,3) (marked in red) needs a Conv between the sliding window $\nabla m_t^{l+1}[2:3, 3:4]$ and the weight kernel. Given the binary membrane potential gradient mask ($\nabla \tilde{m}_t^{l+1}$), we use a Tag to indicate the state (valid/invalid) of the elements in the sliding window. Next, the Priority Encoder produces a valid Conv ID per cycle based on the Tag. The valid weight value can be found through wID which corresponds to the valid Conv ID and ∇mID can be easily acquired based on the Conv ID and ∇sID in ∇mID Decoder.

Next, PE units perform the valid MACs. Figure 3.6(c) shows an example of the workflow in one PE. Each PE in a PE group executes the MACs according to one of the tasks in the corresponding Effectual I∩O Finder. The weight from w^l and the input from ∇m_t^{l+1} are read according to wID and ∇mID, respectively; the MAC result is written into the partial sum (ps_t^l) buffer. In real implementation, PEs in the same PE group write the result to independent partial sum buffers. The accumulated results in a tile will be reordered in the Acc unit. Different PEs in the same PE group reuse the weight buffer. After all PE groups complete the Conv of a tile, PE groups in the same column send their calculated partial sums to the Acc unit for accumulation. Note that, although we have balanced the amount of ∇sIDs across the Output ID Buffers, the amount of valid Conv IDs associated with each Output ID Buffer still varies due to the different input sparsity. Thus, PEs in the PE array work asynchronously during execution but synchronize after all PE groups complete the computation of a tile.

At last, we calculate ∇s_t^l and ∇m_t^l in the Grad unit. As in Equation (2.4)-(2.5), we split the gradient calculation into two phases. In phase 1, Grad takes ps_t^l, m_t^l and ∇m_t^{l+1} to generate ∇s_t^l. In phase 2, besides the calculation of ∇m_t^l, we need to generate the membrane potential gradient mask $\nabla \tilde{m}_t^l$ that reflects the input sparsity of the next backpropagated layer. These two phases can be easily implemented with element-wise operations.

For FC layers, we do not consider any sparsity, since the computation workloads in FC layers are much fewer than those in Conv layers. We disable Effectual O Finders and Effectual I∩O Finders when performing FC layers. In the PE array, the weight buffers in the same column store the weights of different ∇m_t^{l+1} channels but of the same ∇s_t^l channel. The pooling layer can be easily integrated into the Grad unit.

3.2.4 Overall Architecture and Pipeline

Overall Architecture

Figure 3.7 shows the overall architecture of *H2Learn*. In order to reduce the conflicts in data load and store, we use three external memory spaces for simplicity. Specifically, Mem 0 and Mem 1 are used to store spikes, membrane potentials, membrane potential gradients, gradient masks, and weights. During the current forward pass, Forward Engine writes the results into Mem 0(1); while in the next forward pass, the results will be alternatively written into Mem 1(0). Weight Update Engine and Backward Engine request the saved data in the forward pass as their inputs. Mem 2 is designed to store weight gradients that are only used by Weight Update Engine. In real implementation, a single external memory space with a high-bandwidth arbiter is also a possible solution.

We consider the scalability of each processing engine in two directions. The first direction is to increase the size of PE arrays. However, with this change, we should also increase the capacity of global buffers and the off-chip memory bandwidth. Notice that the numbers of rows and columns in each PE array correspond to the number of FMs and timesteps in Conv layers, thus a too large PE array size will decrease the resource utilization for a given Conv layer. Another direction is to increase the number of compute units in a PE group and the number of Acc units. Specifically, in Forward Engine and

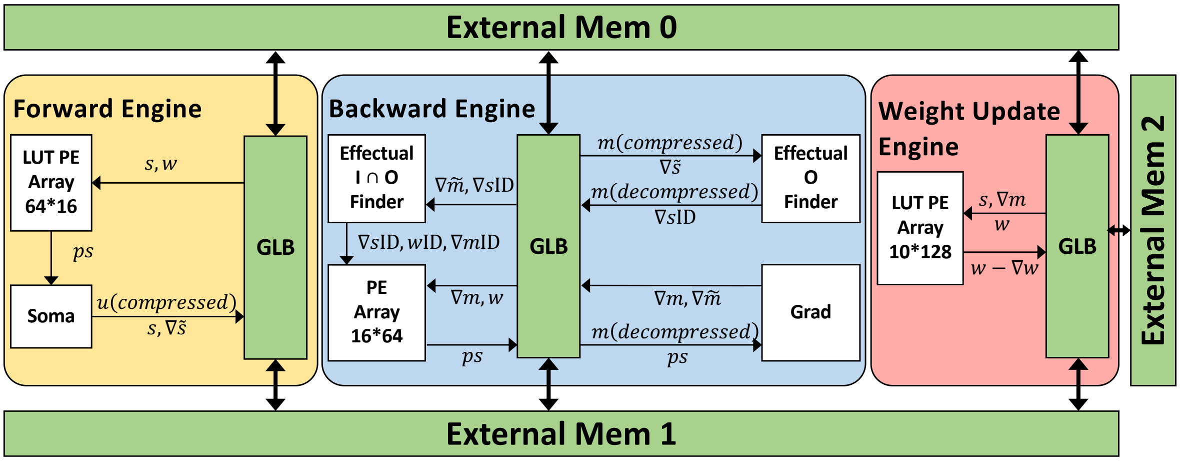


Figure 3.7: Overall architecture of *H2Learn*.

Weight Update Engine, we increase the number of Acc units to process multiple tiles simultaneously; in Backward Engine, we increase both the number of compute units in a PE group and the number of Acc units. Besides the above scalability directions, we can further use multiple *H2Learn*s to build a distributed system. Each of them runs the learning algorithm with different input samples, and the weight gradients are gathered after all of them finish training.

Dataflow for One Layer during Training

The workloads for different engines in *H2Learn* are shown in Figure 3.2, Figure 3.4, and Figure3.5. In order to process a network layer under different setting, each engine adopt a specific dataflow. We can use four compute dimensions to describe the high-level dataflow: 1. BT (batch and time step); 2. HW (height and width); 3. C^l (input channel); 4. C^{l+1} (output channel). The dataflows for each engine are shown in Figure 3.8. Forward Engine and Backward Engine involve neuronal dynamics units. We expect the neuron dynamics engines can work in pipeline with the PE array. Therefore for these two engines we first go through the BT dimension, since the neuronal dynamics has data

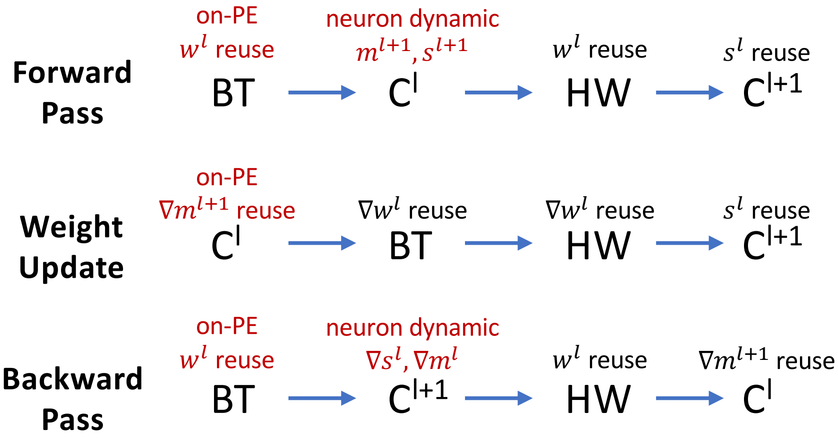


Figure 3.8: Dataflows of different engines for one SNN layer. The execution flow can be decoupled into BT (batch and timestep), HW (height and width), C^l (input channel), and C^{l+1} (output channel) dimensions.

dependency on the time step. Then, we go through C^l and C^{l+1} dimensions for Forward Engine and Backward Engine respectively to process neuronal dynamics. Another reason we select the BT dimension first is that weights are stored on-chip, samples from different batches and time steps can reuse the weights. The last two compute dimensions do not have impact on performance. For Weight Update Engine, there is no neuronal dynamics unit, thus the reuse of on-chip data has the highest priority. Since the spike data is in in the binary format with small data volume, it has the lowest reuse priority in our design. Based on the dataflow, $H2Learn$ can process an SNN layer with arbitrary layer settings.

Execution Pipeline

Figure 3.9 shows the execution flow of training. The yellow, blue, and red boxes indicate the execution in Forward Engine, Backward Engine, and Weight Update Engine, respectively. The sub-batch represents the batch size that each engine can process at a time. The batch group represents the number of sub-batches for the update of weights. Although every sub-batch involves the calculation of weight gradients, only the last sub-batch in a batch group triggers the weight update. Therefore, the overall batch size

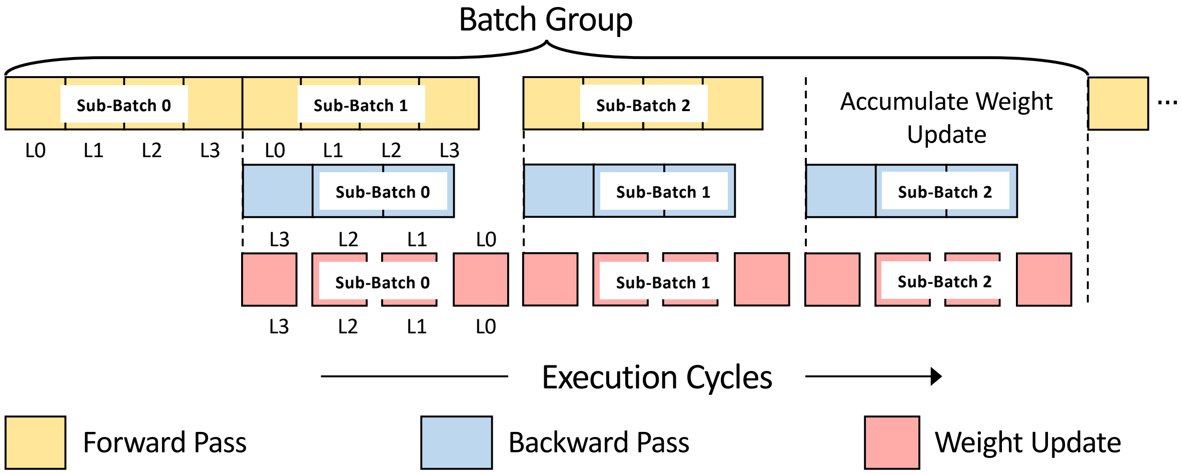


Figure 3.9: Overall pipeline during learning.

can be flexibly reconfigured by adjusting the batch group size. During training, the three engines can be pipelined for higher throughput, i.e., when Backward Engine and Weight Update Engine are processing the current sub-batch, Forward Engine can process the next sub-batch. To enhance the overlap between forward and backward passes and reduce the amount of data need to store in the forward pass, each sub-batch should contain as few samples as possible. We set the sub-batch size to 4 in our design to well utilize the hardware resource.

In addition, some networks involve an encoding layer that takes high-precision inputs instead of binary inputs. We use the Backward Engine in *H2Learn* to process the encoding layer. For the pooling layer without weights, we implement it in the Soma unit rather than the LUT PE.

3.3 Evaluation

3.3.1 Experimental Setup

Our experiments focus on pattern recognition tasks in both image and spike based datasets that are widely used for SNN evaluation. The image-based datasets include MNIST [79], CIFAR10 [80] and ImageNet [81] that are sampled to spikes; while the spike-based datasets include N-MNIST [82] and CIFAR10-DVS [83] that are originally acquired through DVS [84]. The network configurations are detailed in Table 3.3.1.

Dataset	Input Size	Network Structure
N-MNIST	$32 * 32 * 2 * T$	128C3-128C3-AP2-384C3-384C3-AP2-512FC-512FC-10FC
CIFAR10-DVS	$42 * 42 * 2 * T$	
MNIST	$28 * 28 * 1 * T$	64C3(Encoding)-128C3-AP2-256C3-256C3-AP2-512C3-512C3-512FC-512FC-10FC
CIFAR10	$32 * 32 * 3 * T$	
ImageNet	$224 * 224 * T$	64C3S2(Encoding)-128C3S2-256C3S2-256C3S2-384C3-256C3-256C3S2-4096FC-4096FC-1000FC

Table 3.4: Network configurations. T is set to 10.

We build cycle accurate simulators for *H2Learn* and the accelerator baselines in our experiments. The area and energy are measured through synthesized implementations. We implement *H2Learn*'s RTL and synthesize it in Synopsis Design Compiler with TSMC 28nm library. The area and energy of GLBs are estimated via Cacti [85]. In our simulation, we evaluate the average energy per operation for all compute and storage units, and the total energy is obtained by estimating the number of operations.

The configuration of all engines in *H2Learn* are listed in Table3.3.1. In our evaluation, we build a different baseline model for each engine. Specifically, for Forward Engine and Weight Update Engine, the baseline models adopt non-LUT implementation which consume 36,864 and 40,960 adders, under the parallelism of 4. For the baseline model of Backward Engine, the Effectual O Finder and Effectual I∩O Finder units are removed

Forward Engine	LUT PE Array Size	64 (rows) × 16 (cols)
	LUT PE	3 Sub-LUTs/PE, 16 Bytes/Sub-LUT
	Acc	3×64×16 (3072) adders, parallism=4
	# Somas	16
	GLB	503 KB
	Area	21.61 mm^2
Weight Updte Engine	LUT PE Array Size	10 (rows) × 128 (cols)
	LUT PE	2 Sub-LUTs/PE, 32 Bytes/Sub-LUT
	Acc	2×10×128 (2560) adders, parallism=4
	GLB	2684 KB
	Area	22.68 mm^2
Backward Engine	PE Group Array Size	16 (rows) × 64 (cols)
	PE Group Size	4
	# Effectual O Finders	64
	# Effectual I∩O Finders	16×64×4
	PE	16×64×4 MAC
	Acc	16×64×4 adders
	# Grads	64
	GLB	4840.5 KB
	Area	66.17 mm^2

Table 3.5: Specifications of engines in *H2Learn*.

and we do not exploit any sparsity.

Since *H2Learn* focuses on the training scenario of SNNs, our final target comparison platform is the modern GPU, which is the backbone hardware for learning. The baseline GPU version of BPTT-based SNN learning is realized through an open-source Pytorch implementation without CUDA optimization [6]. Table 3.3.1 compares the overall specifications between *H2Learn* and NVIDIA V100 GPU [86]. The power consumption for *H2Learn* can be acquired by evaluating the energy consumption and the training latency for *H2Learn*. We took the max power consumption during SNN learning on *H2Learn*. We will demonstrate that *H2Learn* can achieve substantial speedup and energy efficiency improvement with far less area consumption in Section 3.3.3.

	H2Learn	***GPU V100***
Technology	TSMC 28 nm	TSMC 12 nm
Area	110.46 mm^2	815 mm^2
Clock Frequency	800 MHz	1530 MHZ
Off-chip Memory Bandwidth	128 GB/s×3	900 GB/s
Throughput	27.85 TFLOPS	15.7 TFLOPS
Power	20.57 W	300 W

Table 3.6: Specifications of *H2Learn* and NVIDIA V100 GPU.

3.3.2 Evaluation of Engines in H2Learn

Forward Engine & Weight Update Engine

Now, we evaluate the LUT-based Engines, i.e., Forward Engine and Weight Update Engine. Fig. 3.10 shows how the PE configuration affects the area and energy consumption, where both PEs and Acc units are considered. The baseline is a non-LUT design. The LUT configuration is determined by the number of sub-LUTs per PE (#sub-LUTs/PE) and the size of each sub-LUT (sub-LUT size). We assume that the size of each sliding window in Conv is 3×3. We find that when we decrease the #sub-LUTs/PE, the area and energy of computation units (i.e., adders) are reduced, however, the requirement for register files is increased. This is actually a trade-off, i.e., fewer sub-LUTs per PE can save more adders for compute but require more register files for storage.

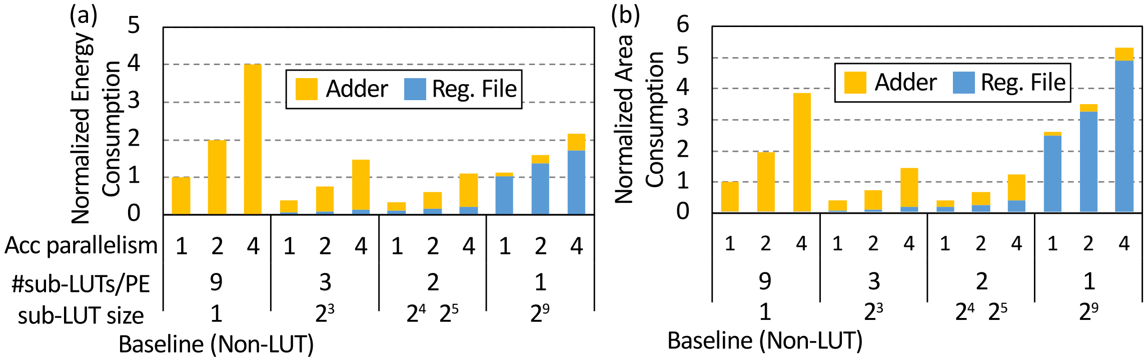


Figure 3.10: Evaluation of LUT PE in Forward Engine under different configurations: (a) area and (b) energy.

We also estimate the area and energy consumption with different parallelism settings in Acc units. Here the parallelism means that multiple tiles are simultaneously processed in the LUT PE array and the resources for Acc units are copied accordingly. From the results, it can be seen, besides the increased adders in Acc units, the requirement for register files is also increased with a slower slope, since we do not scale up the number of sub-LUTs but use MUX to make sure multiple elements can be read from each sub-LUT in the meantime.

In our design, considering the unified sub-LUT size and lower overhead, we set the #sub-LUTs/PE to 3 and the sub-LUT size to 8 in Forward Engine, corresponding to 3×3 sliding windows. In Weigh Update Engine, the #sub-LUTs/PE is 2 and the sub-LUT size is 16, corresponding to 1×8 sliding windows. For a larger Conv kernel size, we can partition it into multiple smaller kernels and map onto multiple LUT PEs. The overall LUT sizes in Forward Engine and Weight Update Engine are 48 KB and 80 KB, respectively.

Next, we analyze the entire Forward Engine and Weight Update Engine when per-

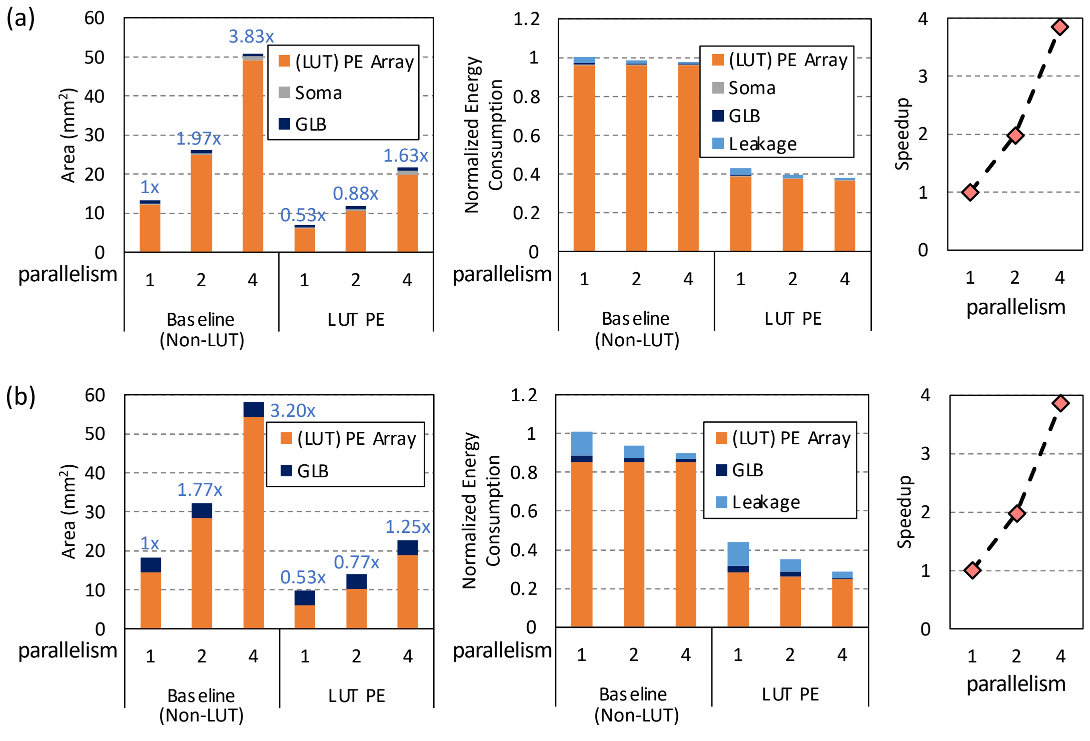


Figure 3.11: Area, energy, and throughput of (a) Forward Engine and (b) Weight Update Engine.

forming a Conv layer. The size of s, m, and ∇m is $4 \times 10 \times 256 \times 56 \times 56$ ($N \times T \times C_s$(or C_m, or $C_{\nabla m}$)$\times W \times H$), and the size of w is $3\times3\times256\times256$ ($k\times k\times C_s\times C_m$(or $C_{\nabla m}$)). The results are shown in Fig. 3.11, where the baseline architecture adopts the naive accumulation-based rather than LUT-based PE. We have the following observations: (1) The PE array consumes most of the area and the LUT-based design can significantly reduce the area overhead; (2) Although the outputs of both Forward Engine and Weight Update Engine are in FP16, the number of columns in the PE array of Weight Update Engine is much larger, leading to a larger GLB size; (3) The energy consumption with different parallelism setting is close, because the amount of total workloads under different parallelism is identical; (4) The throughput can be improved as the parallelism increases and the leakage energy can be reduced; (5) Compared to the baseline architecture when the parallelism equals 4, our LUT-based solution can achieve 2.35× area saving, 2.58×

energy saving in Forward Engine and 2.56× area saving, 3.00× energy saving in Weight Update Engine. Notice that the partial sums in LUTs need to be reprogrammed by the Acc units when the contents in LUTs need to change. Our dataflow in Fig. 3.8 can help increase the reusability of the data in LUTs. Specifically, in Forward Engine, the sliding windows along the height/width of a tile, batch, and timestep dimensions share the same weights; in Weight Update Engine, the sliding windows share the same gradients along the input channel dimension. Therefore, LUTs can be reused a lot of times once reprogrammed, and the LUT reprogramming overhead iss negligible.

Backward Engine

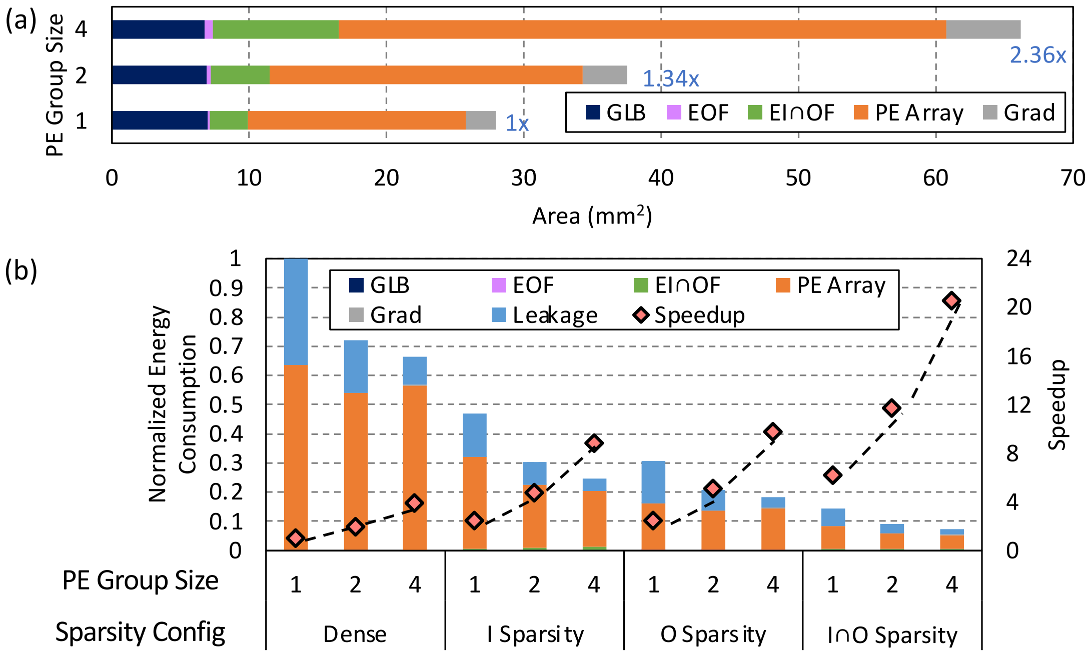


Figure 3.12: Backward Engine evaluation: (a) area overhead; (b) energy consumption and speedup.

We adopt the same Conv layer as we used in Section 3.3.2. As depicted in Fig. 3.12(a), as the PE group size grows, the area overhead increases but the GLB size does not change obviously since the resulting output volume keeps the same. In Fig. 3.12(b), we measure the energy and throughput, where we set both the sparsity of $\nabla\tilde{s}^l$ (output sparsity) and $\nabla\tilde{m}^{l+1}$ (input sparsity) to 75%. We first build a dense baseline model without considering any input and output sparsity. We also adopt our architecture to exploit only input or output sparsity as two other baselines. From the results, we find that the leakage consumes a huge amount of energy, which mainly comes from GLB. Another observation is that the energy consumption of the PE array (including the Acc units) occupies the most in the dense architecture, because it cannot bypass any computation. Also, the energy consumed by PE array is much higher when we consider the input sparsity only, since more accumulations of the partial sums are needed when compared with those considering the output sparsity. Since the sparsity settings of $\nabla\tilde{s}^l$ and $\nabla\tilde{m}^{l+1}$ are the same, the speedup results are similar when we consider the input or output sparsity only. Finally, when we consider both the input and output sparsity , we can achieve 5.19× speedup and 9.24× energy saving compared with the dense baseline architecture.

Design Space Analysis

Table 3.3.1 shows the configurations of *H2Learn*. We adopt output stationary dataflow in all engines. Since the inputs of Forward Engine are binary spikes that are more compact than the outputs, we set a larger number of rows (64) in the PE array. Because of the output stationary dataflow, the result is written to external memory for every $C_s/64$ grid iterations, which can help reduce the data traffic. Note that we use ping-pong buffer in GLBs. In Backward Engine, both inputs and outputs are in the FP16 format. We shrink the number of rows (16) but increase the number of columns (64), such that the amount of inputs needing to feed is reduced and the outputs can take longer time ($C_{\nabla m}/16$ grid iterations) to be written to external memory. For Weight Update Engine, the number of rows is set to 10 (allowing to deal with 10 timesteps during a grid iteration), and the number of columns (128) is selected according to the PE array sizes in other two engines.

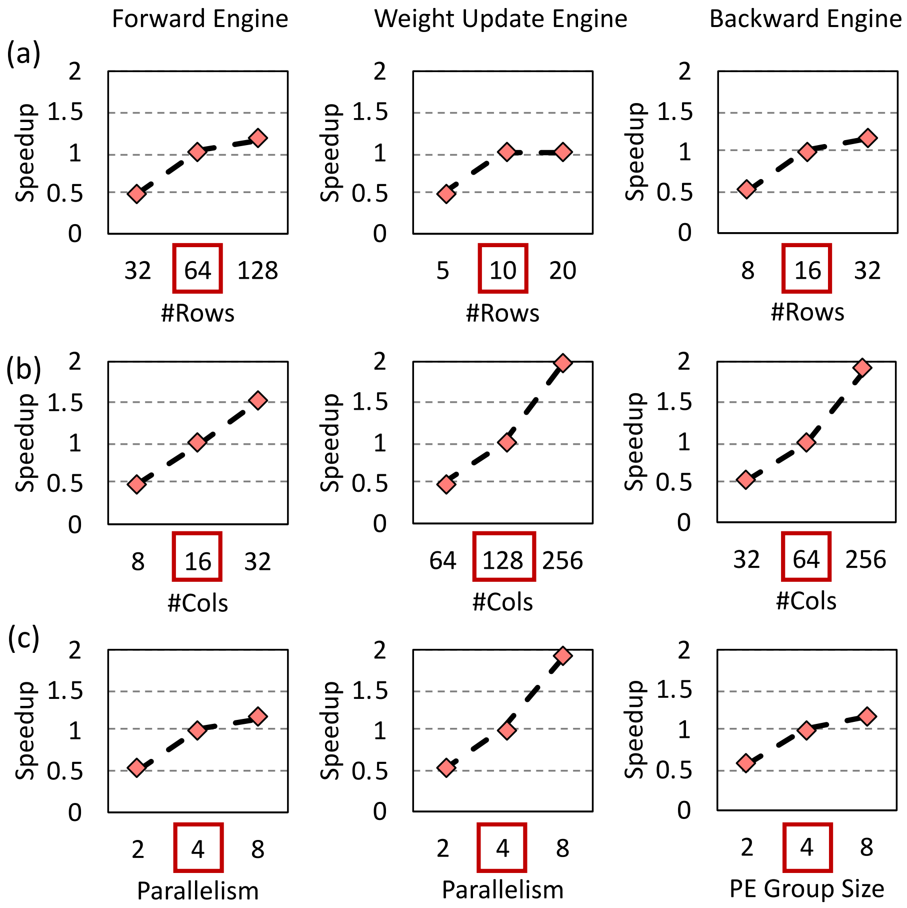


Figure 3.13: Evaluation of engines in *H2Learn* with different (a) number of PE array rows, (b) number of PE array columns, and (c) parallelism.

Fig. 3.13 estimates the throughput of each engine in *H2Learn* with different architecture configurations, including the number of PE array rows, columns, and the parallelism. The configurations within the red boxes are our optimal selections for each engine in Table 3.3.1, which consider both optimal performance and balanced compute resources in different engines. The optimal selection can fully utilize but will not be blocked by the off-chip memory bandwidth. Notice that when we evaluate one architecture configuration, we will fix the other two at the optimal settings. We also adopt the same Conv layer for evaluation as in Section 3.3.2. From the results, we find that the optimal settings can always gain a 2× speedup when compared with the corresponding halved settings. This implies that the external memory bandwidth can satisfy our optimal settings. However, if we keep scaling up the PE array size or the parallelism, the throughput gain would degrade. In Forward Engine, all architecture configurations cannot get another 2× speedup by doubling the optimal settings. In Weight Update Engine, each PE array row deals with a unique timestep, thus the increase of extra rows is useless if the number of timesteps is small; however, the throughput can be doubled when we scale up the PE array columns or the parallelism. The reason is that, the final outputs of Weight Update Engine is the weight gradients which have a small volume and can stay on-chip until the entire Conv across all FMs finished. This lowers the bandwidth requirement and leaves room for the increase of compute resources. For Backward Engine, the inputs (i.e., ∇u) become the key factor to determine the external memory bandwidth requirement, because the input data type is FP16 and the load of inputs across all channels is more frequently than the write of stationary outputs. The further increase of PE array rows and PE group size cannot achieve 2× performance gain due to the memory bandwidth limitation. In contrast, when the number of PE array columns is doubled, 2× speedup can be obtained, since the amount of input loads is unchanged.

3.3.3 Comparison with SpinalFlow and GPU

We compare *H2Learn* with a state-of-the-art SNN inference accelerator *SpinalFlow* [36] and NVIDIA V100 GPU [86] on CIFAR10. The input and output sparsity of the networks are shown in Table 3.1.3. Since *SpinalFlow* only supports inference, in Fig. 3.14(a), we compare our Forward Engine with it. We find that *H2Learn* achieves improvement in terms of area and energy. Although *SpinalFlow* skips the computations with zero inputs, they need to store entire weights of all output channels to perform computations associated with a valid input, which consumes large area for weight storage and significant power consumption for data accesses. Our *H2Learn* still gets improvement via the LUT-based design to fuse accumulations of multiple input points. From the functionality perspective, *H2Learn* shows three distinctive characteristics: 1. *H2Learn* targets learning while *SpinalFlow* focuses on inference; 2. *H2Learn* does not have restrictions on coding schemes, while *SpinalFlow* only supports temporal coding; 3. *H2Learn* can support the first encoding layer with hybrid data formats while *SpinalFlow* cannot. Also, the SpinalFlow cannot support the encoding layer for image-based datasets with continuous inputs, which can be easily handled by Backward Engine in *H2Learn*.

Then, we compare the throughput of engines in *H2Learn* with NVIDIA V100 GPU in Fig.3.14(b). We implement the GPU version of SNN learning in Pytorch. Different from the sub-batch-wise pipeline in H2Learn as Fig.3.9, the forward pass and backward pass (along with weight update) are performed sequentially at the grain of the whole batch on GPUs as common handling. We find that *H2Learn* achieves speedup especially in early layers during weight update. In shallow layers, the FM sizes are larger but the number of channels is smaller; besides, weight update needs a 4D rather than 2D Conv. GPU might be inefficient to handle these situations.

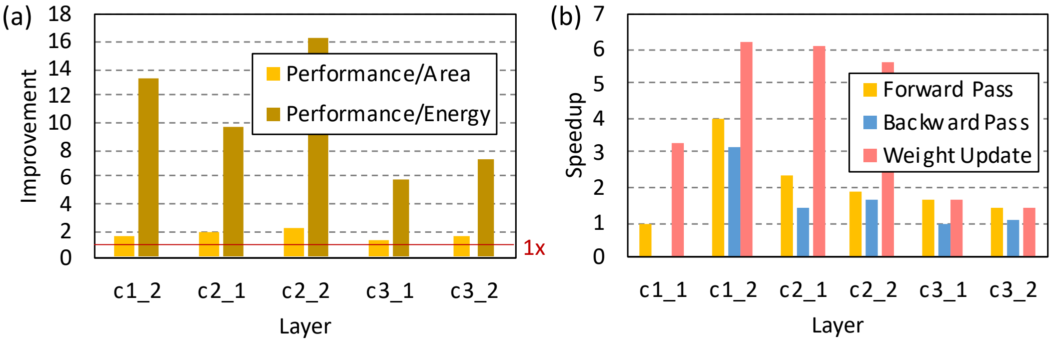


Figure 3.14: Evaluation of *H2Learn* on CIFAR10: (a) Forward Engine compared with SpinalFlow [36] SNN inference accelerator; (b) throughput of engines compared with NVIDIA V100 GPU.

Fig. 3.15 shows the comparison between *H2Learn* and NVIDIA V100 GPU during training. Because we focus on the processor design and do not estimate the power of the off-chip memory, here we exclude the HBM power of GPU for fairness. Among the results, *H2Learn* achieve 5.74-10.20× speedup and 5.25-7.12× energy saving. We find that *H2Learn* takes more benefits on the ImageNet dataset. The potential reason might be caused by more data preprocessing on GPUs under a large FM size. At last, *H2Learn* is 7.38× more efficient in area overhead.

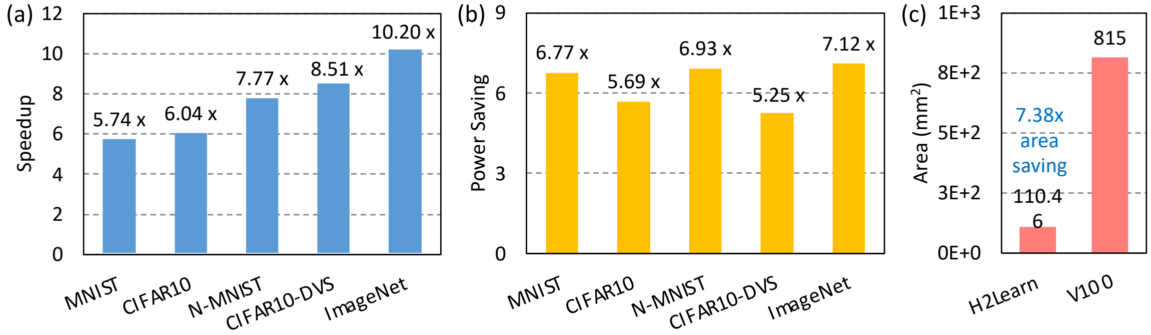


Figure 3.15: Comparison with NVIDIA V100 GPU in terms of (a) throughput, (b) power, and (c) area.

3.3.4 Comparison with Loihi and FPGA

Model	32C3-AP2-64C3-AP2-128C3-AP2-512FC-10FC	
Plateform	Loihi	H2Learn
Latency/sample (s)	1.02×10^{-3}	1.15×10^{-4}
Energy/sample (J)	6.60×10^{-5}	3.33×10^{-4}
Model	64C5(Encoding)-AP2-64C5-AP2-128FC-10FC	
Plateform	FPGA	H2Learn
Latency/sample (s)	6.11×10^{-3}	1.21×10^{-2}
Energy/sample (J)	3.00×10^{-2}	3.58×10^{-4}

Table 3.7: Comparison with Loihi [9] and FPGA [87] for SNN inference. Here we set $T = 10$ and the batch size to 1.

Except for the above baseline platforms such as SpinalFlow and GPU, researchers also design neuromorphic chips to support both inference and learning for SNNs. Currently, Loihi [9, 88] is the most advanced and mature chip that can support unsupervised SNN learning. One major reason that Loihi cannot directly implement BPTT-based SNN training is that it is unable to store and load the huge intermediate data (e.g., membrane potentials and spikes across all layers and all timesteps) during training. Thus, we compare SNN inference between *H2Learn* and Loihi in Table 3.3.4. Here we get the network model and performance on Loihi from reported data [89]. We evaluate the SNN inference on H2Learn under the same network structure. From the results, the Forward Engine in *H2Learn* can achieve 8.87× speedup but consume 5.06× more energy compared to Loihi. Since Loihi adopts an event driven implementation, the energy efficiency is better than *H2Learn*.

We further compare the performance of SNN inference between *H2Learn* and an FPGA implementation [87] in Table 3.3.4. From the results, *H2Learn* achieves 50.50× speedup and 83.79× energy saving. The main reasons that FPGA suffers a degraded performance are due to the limited on-chip resources and low clock frequency.

3.4 Related Work

3.4.1 SNN Learning Algorithms

Besides the BPTT-based SNN learning adopted in this work, there is another family of learning methods named ANN-to-SNN conversion [90, 91, 92, 93]. These learning algorithms convert a pretrained ANN model to its SNN counterpart sharing the same network structure. ANN-to-SNN conversion can achieve slightly better accuracy compared to the direct learning with BPTT. However, ANN-to-SNN conversion methods usually need much more timesteps to maintain accuracy, resulting in longer execution latency and higher energy consumption during the inference stage. In addition, some studies also design specific learning algorithms [94, 95] to achieve higher accuracy. Although additional circuits and dataflow design are required to support these learning methods on $H2Learn$, we have solved most costly operations during training.

3.4.2 LUT in Neuromorphic Chips

In $H2Learn$, LUTs are exploited to calculate the partial sums of potentials. Some other accelerators also adopt LUTs for SNN inference or training. Specifically, J. Pu et al. [96] used LUTs to store the connections between neurons rather than to do computation. SPARE [97] accelerates the high-order polynomials and transcendental functions in STDP training with LUTs. In contrast, the functionality of LUTs in *H2Learn* is distinct from these works.

3.5 Conclusion

This Chapter introduces *H2Learn*, an end-to-end accelerator that can implement BPTT-based SNN learning for both high accuracy and high efficiency. The LUT-based PE design in Forward Engine and Weight Update Engine exploits the spike-based computation; the dual-sparsity-aware Backward Engine exploits both input and output sparsity. Compared with the modern NVIDIA V100 GPU, *H2Learn* demonstrates 7.38× area saving, 5.74-10.20× speedup, and 5.25-7.12× power saving on several typical benchmark datasets.

Chapter 4

Accelerating Spatiotemporal Supervised Training of Large-Scale Spiking Neural Networks on GPU

Previous Chapter provides an accelerator solution to improve the efficiency of BPTT-based SNN training. However, training on GPUs still remains inefficient due to the complex spatiotemporal dynamics and huge memory consumption, which restricts the model exploration for SNNs and prevents the advance of neuromorphic computing.

In this chapter, a framework is introduced to solve the inefficiency of BPTT-based SNN training on modern GPUs. To reduce the memory consumption, the dataflow is optimized by saving CONV/FC results only in the forward pass and recomputing other intermediate results in the backward pass. Then, we customize kernel functions to accelerate the neural dynamics for all training stages. Finally, a Pytorch interface is provided to make our framework easy-to-deploy in real systems. Compared to vanilla Pytorch implementation, the proposed framework can achieve up to 2.13× end-to-end speedup and consume only 0.41× peak memory on the CIFAR10 dataset. Moreover, for the distributed training on the large ImageNet dataset, we can achieve up to 1.81× end-to-end speedup and consume only 0.38× peak memory.

4.1 Overview and Motivation

Although BPTT-based training algorithms boost the accuracy of SNNs, it is inefficient to train an SNN model on GPUs. Compared to the well-optimized training of an artificial neural network (ANN) model on GPU, training an SNN model with BPTT-based algorithms would encounter two challenges: (1) there are more intermediate data; (2) the spatiotemporal dynamics and computational operations are more complex. The first challenge increases the memory consumption, which limits the exploration space of SNN models given the same hardware resources. The second challenge causes frequent GPU kernel launching that dramatically degrades the execution performance. Some researches explore the optimization of *CUDA* codes on GPU for other applications such as graph neural network [98], and Transformer [99]. These studies identify distinct operations for each application, their optimization methods cannot be directly exploited in BPTT-based SNN learning.

4.1.1 SNN Model

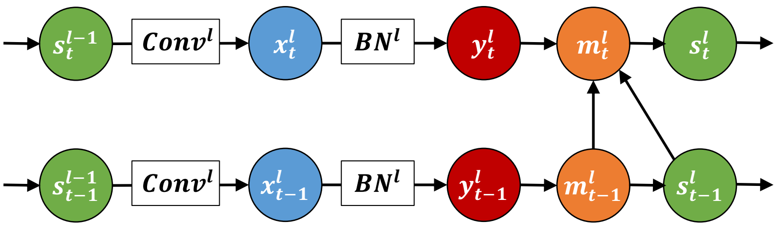


Figure 4.1: Illustration of the "Conv→BN→LIF" information flow in the FP stage.

The SNN models used in this Chapter involve an additional batch-normalization operation in each layer. The dataflow of one SNN layer is shown in Figure 4.1. Sepcifically, the membrane potential update in Equation 2.2 is substituted with

$$\boldsymbol{m}_t^l = \underbrace{\alpha \cdot \boldsymbol{m}_{t-1}^l \cdot (1 - \boldsymbol{s}_{t-1}^l)}_{temporal} + \underbrace{\boldsymbol{y}_t^l}_{spatial}, \tag{4.1}$$

where the spatial part $\boldsymbol{y}$ is the result after the BN operation, as in Figure 4.1. The BN follows

$$\boldsymbol{y}_t^l[j] = \frac{\boldsymbol{x}_t^l[j] - \mu^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}} \gamma^l[j] + \beta^l[j], \tag{4.2}$$

where $\boldsymbol{x}_t^l[j]$ and $\boldsymbol{y}_t^l[j]$ stand for Conv/Pool/FC and BN results of the j-th channel in the l-th layer at the t-th time step, respectively. The BN [100] is achieved by subtracting the mean $\mu^l[j]$ and dividing the standard deviation $\sqrt{(\sigma^l[j])^2 + \epsilon}$, where $\mu^l[j]$ and $\sigma^l[j]$ are the results from all elements in the j-th channel of $\boldsymbol{x}^l$ across all samples in a batch and time steps. ϵ is a small value to avoid the division error. γ and β are two trainable parameters shared by all neurons in the same channel. In Equation 4.2, $\boldsymbol{x}$ can be the spatial result from a Conv/Pool/FC layer. Taking the Conv result as an example, it is calculated through Equation 2.3.

During the backward propagation, the gradient of spike $\nabla \boldsymbol{s}_t^l$ and membrane potential $\nabla \boldsymbol{m}_t^l$ can be computed following Equation 2.4-2.5. From Equation 4.1, it is easy to find that $\nabla \boldsymbol{m}_t^l = \nabla \boldsymbol{y}_t^l$. Based on $\nabla \boldsymbol{y}_t^l$, we can get the gradient of the Conv result $\nabla \boldsymbol{x}_t^l$ through

$$\nabla \boldsymbol{x}_t^l[j] = \frac{\gamma^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}} \left(\nabla \boldsymbol{y}_t^l[j] - \frac{\boldsymbol{I}}{n} \sum_{t,n_j} \nabla y_t^l[n_j] - \frac{\hat{\boldsymbol{x}}_t^l[j]}{n} \sum_{t,n_j} \nabla y_t^l[n_j] \hat{x}_t^l[n_j] \right), \tag{4.3}$$

$$\hat{\boldsymbol{x}}_t^l[j] = \frac{\boldsymbol{x}_t^l[j] - \mu^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}}, \tag{4.4}$$

where $\nabla \boldsymbol{x}_t^l$ is comprised of the partial derivative of $\boldsymbol{y}_t^l$ with respect to $\boldsymbol{x}_t^l$, $\boldsymbol{\mu}^l$, and $\boldsymbol{\sigma}^l$ in

Equation 4.2, which corresponds to the three terms in the brackets of Equation 4.3. Since the neurons in the same channel share identical mean and standard deviation values, n_j indicates the neurons in the j^{th} channel and n denotes the total number of neurons in the j^{th} channel.

In the PU stage, we first update the trainable parameters of the BN operation following

$$\nabla\gamma^l[j] = \sum\nolimits_{t,n_j} \nabla y_t^l[n_j]\hat{x}_t^l[n_j], \tag{4.5}$$

$$\nabla\beta^l[j] = \sum\nolimits_{t,n_j} \nabla y_t^l[n_j]. \tag{4.6}$$

The above parameter update is derived from Equation 4.2.

4.1.2 Motivation

Currently, most of the BPTT-based SNN studies are simulated on GPU with the Pytorch programming environment [20, 100]. We use the vanilla Pytorch implementation [100] to characterize the training performance. Figure 4.2 presents the latency and peak memory consumption for training an SNN model with one epoch on the CIFAR10 dataset under different model settings. The network size is medium and the structure is provided in Table 4.1. The memory consumption for feature maps (i.e., $\boldsymbol{s}$, $\boldsymbol{m}$, $\boldsymbol{x}$, $\boldsymbol{y}$) at each time step is 71.13 MB. $Net.\ with\ LIF + BN$ and $Net.\ with\ LIF$ denote the networks with and without BN. In order to estimate the impact of the $fire$ function and the membrane potential calculation (i.e., LIF dynamics), we additionally design a network with only Conv/FC layers, which is denoted as *Baseline*. The *Baseline* network is functionally incorrect and we just use it for comparison.

Figure 4.2(a) compares the peak memory consumption during training. Since the

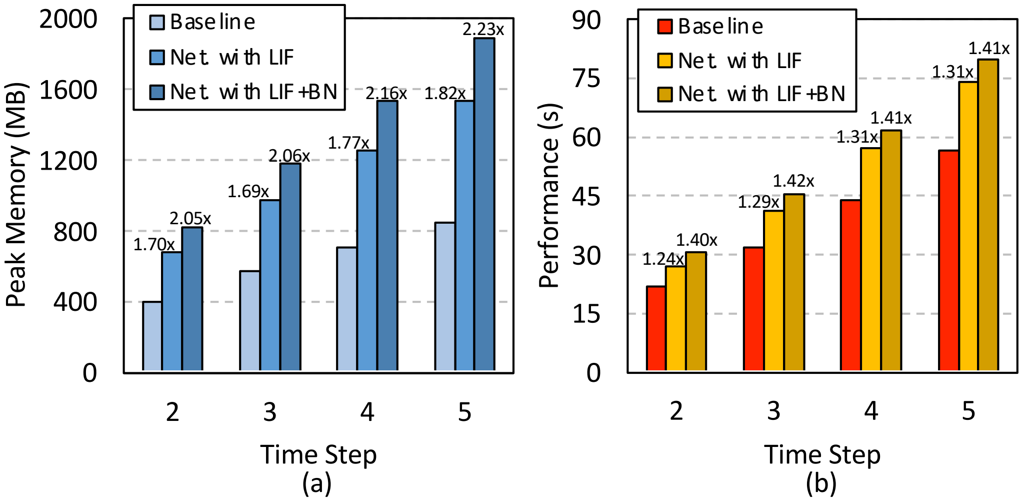


Figure 4.2: Characterization of the vanilla Pytorch implementation of SNN training with one epoch on CIFAR10: (a) peak memory consumption; (b) latency.

peak memory consumption determines how many GPUs are needed, a lower memory consumption can help reduce the resource overhead. It can be seen that incorporating the LIF operation would increase the peak memory consumption by 1.69 ∼ 1.82×. The increased memory consumption comes from $\boldsymbol{u}$ and $\boldsymbol{s}$. The further incorporation of BN consumes 2.05 ∼ 2.23× peak memory compared to the baseline, due to the extra storage of $\boldsymbol{y}$. **We find that the intermediate data consume lots of memory, which finally impacts design space exploration for SNN models**. Figure 4.2(b) further compares the training latency. Compared to the baseline, the LIF dynamics increases the training latency by 1.24 ∼ 1.31×; the further incorporation of BN consumes 1.40 ∼ 1.42× latency compared to the baseline. **From the result, the computation latency for the operations besides Conv/FC cannot be ignored.**

Given the above observations, we find that the optimization of LIF and BN operations during SNN training is valuable.

4.2 Optimization on GPU

In this section we first present how to optimize the BPTT-based SNN training dataflow for saving the storage consumption. Then we detail our GPU kernel design to fuse arithmetic operations for eliminating frequent kernel launching.

4.2.1 Dataflow Optimization

In SNN training, many intermediate variables are generated and stored in the FP stage, which are accessed in BP and PU stages. Whereas, this causes huge memory consumption. In order to reduce the memory overhead, we propose to only store parts of intermediate data in the FP stage, and recompute the missing ones in BP and PU stages.

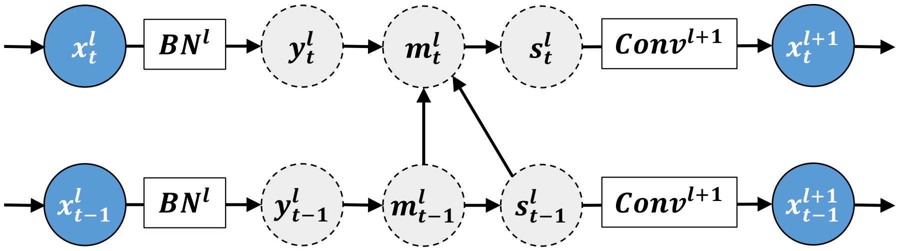


Figure 4.3: The optimized information flow that only stores $\boldsymbol{x}$ in the FP stage.

With the above idea, we choose to save Conv (or FC) results (i.e., $\boldsymbol{x}$) as in Figure 4.3. In this example, BN results, membrane potentials, and spike events are all abandoned in the FP stage. The reason that we select Conv results to store is that Conv takes most of computations, and thus recomputing Conv results may cause long training latency. Although recomputing other intermediate data needs complex arithmetic computations, all of them are element-wise that occupy fewer workloads.

The goal of BP and PU stages are to calculate the gradients of Conv results (i.e., $\nabla \boldsymbol{x}^l$), the gradients of Conv parameters (i.e., $\nabla \boldsymbol{w}^{l+1}$ & $\nabla \boldsymbol{b}^{l+1}$) and BN parameters (i.e.,

$\nabla\boldsymbol{\gamma}^l$ & $\nabla\boldsymbol{\beta}^l$). Three steps can be summarized in BP and PU stages as Figure 4.4. In the first step, we recompute $\boldsymbol{m}^l$ & $\boldsymbol{s}^l$, which are abandoned in the FP stage, as in Figure 4.4(a). Then, according to the recomputed $\boldsymbol{s}^l$ and the backpropagated $\nabla\boldsymbol{x}^{l+1}$, we get the gradients of Conv parameters in layer $l+1$. Also, we get the spatial part of $\nabla\boldsymbol{s}^l$ in Equation 2.4, as in Figure 4.4(b). In the last step, we compute $\nabla\boldsymbol{x}^l$ and the gradients of BN parameters, as in Figure 4.4(c). The detailed algorithms for each step will be provided in the next subsection.

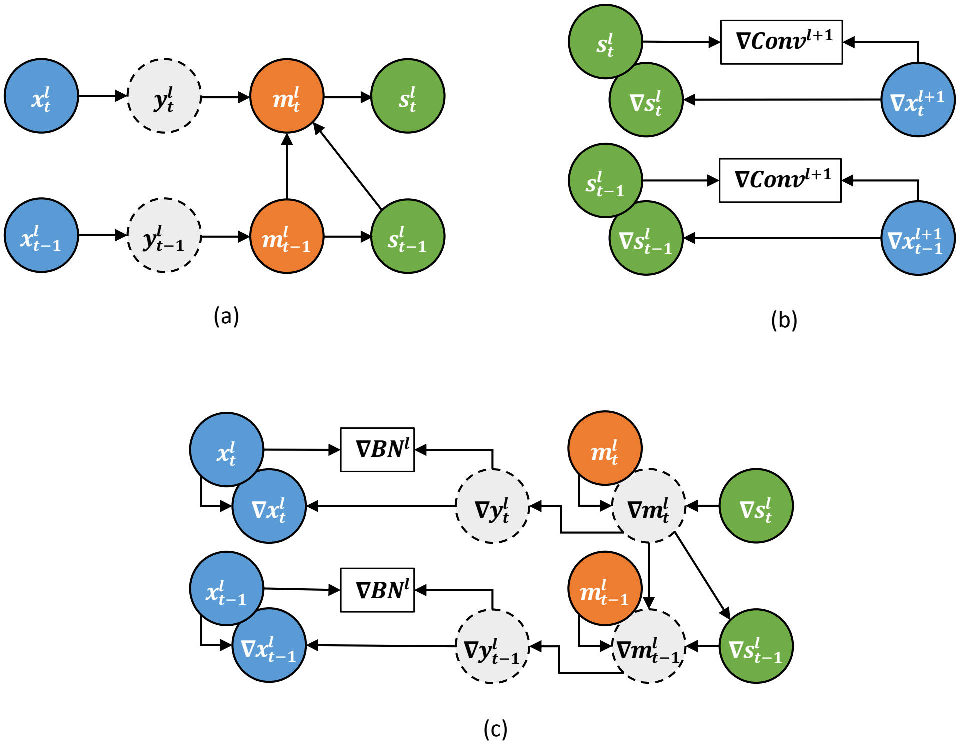


Figure 4.4: The three steps in BP and PU stages: (a) recompute $\boldsymbol{m}^l$ and $\boldsymbol{s}^l$ (Algorithm 1); (b) compute the gradients of Conv parameters (i.e., $\nabla\boldsymbol{w}^l$ & $\nabla\boldsymbol{b}^l$) and the spatial part of $\nabla\boldsymbol{s}^l$ (Algorithm 4); (c) compute the gradients of BN parameters (i.e., $\nabla\boldsymbol{\gamma}^l$ & $\nabla\boldsymbol{\beta}^l$) and the gradients of Conv results (i.e., $\nabla\boldsymbol{x}^l$) (Algorithm 2 & 3).

4.2.2 Kernel Optimization for BP and PU Stages

Except for the huge memory consumption in SNN training, the complex LIF dynamics and BN in the vanilla Pytorch implementation degrade the performance dramatically due to the frequently kernel launching. In this subsection, we explain how to optimize the GPU kernel functions for those operations to accelerate the gradient calculation in BP and PU stages.

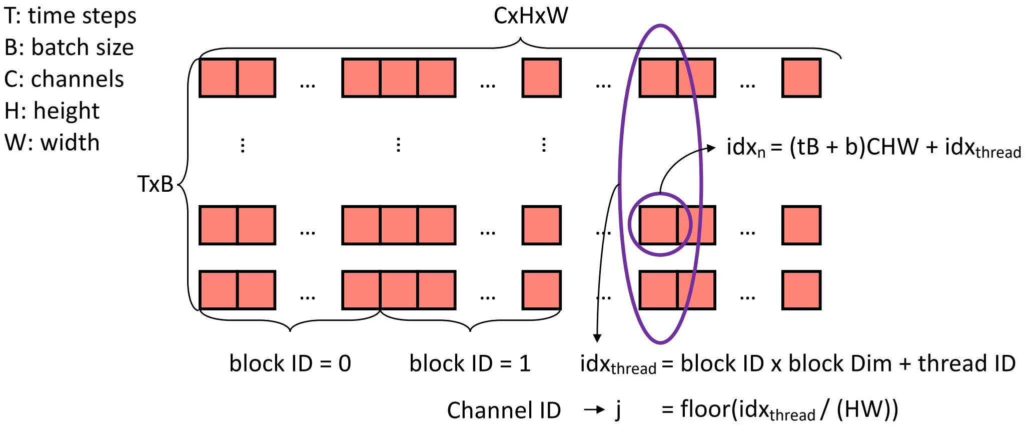

T: time steps
B: batch size
C: channels
H: height
W: width
CxHxW
TxB
idx_n = (tB + b)CHW + idx_{thread}
block ID = 0
block ID = 1
idx_{thread} = block ID x block Dim + thread ID
Channel ID → j = floor(idx_{thread} / (HW))

Figure 4.5: Correspondence between the feature map and threads for the kernel programming. The feature map is stored with the row major order.

Since we abandoned the storage of membrane potentials and spike events in the FP stage, we need to recompute $\boldsymbol{m}^l$ & $\boldsymbol{s}^l$ based on $\boldsymbol{x}^l$ in BP and PU stages as shown in Figure 4.4(a). The detailed kernel function is provided in Algorithm 1. Here, we fuse all arithmetic operations into one kernel function to avoid the frequent kernel launching. The update of membrane potentials and spike events are dominated by Equation 2.1-4.2. The thread model for GPU kernels is presented in Figure 4.5. We first reshape the tensor from the size of $T \times B \times C \times H \times W$ to a 2D format with the size of $TB \times CHW$. With this reshaping, each thread processes $T \times B$ elements and the total number of threads is CHW. During the processing of the kernel function, we need to know the channel ID (i.e., j) for each thread to get the corresponding parameters for BN. We also need to know the neuron's location (i.e., idx_n) in the original tensor to write back the calculated results. In Algorithm 1, each thread calculates $\boldsymbol{m}^l$ and $\boldsymbol{s}^l$ for different time steps and input samples in a batch, in which $\mu^l[j]$, $\sigma^l[j]$, $\gamma^l[j]$, and $\beta^l[j]$ keep unchanged under the same j. During the kernel processing, $\boldsymbol{m}^l$ and $\boldsymbol{s}^l$ are calculated along the time steps from 1 *to* T, since the calculation at the current time step has data dependency on the previous time step according to Equation 2.2. Notice that this fused kernel function can be directly applied in the FP stage to compute the spike events.

Algorithm 1: Fused FP Kernel with BN

```
Data: $\boldsymbol{x} \in \mathbb{R}^{T\times B\times C\times H\times W}$; $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, $\boldsymbol{\gamma}$, $\boldsymbol{\beta} \in \mathbb{R}^{C}$;
Result: $\boldsymbol{m}$, $\boldsymbol{s} \in \mathbb{R}^{T\times B\times C\times H\times W}$;
begin
    $\mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$; $\gamma_{tmp} = \gamma[j]$; $\beta_{tmp} = \beta[j]$;
    for $b \leftarrow 1$ to $B$ do
        $u_{pre} = 0$; $s_{pre} = 0$;
        for $t \leftarrow 1$ to $T$ do
            // Eq. (2.2) & (4.2)
            $m_{pre} = \alpha m_{pre}(1 - s_{pre}) + \frac{x[idx_n] - \mu_{tmp}}{\sqrt{\sigma_{tmp}^2 + \epsilon}}\gamma_{tmp} + \beta_{tmp}$;
            // Eq. (2.1)
            $s_{pre} = m_{pre} \geq th_f$;
            $m[idx_n] = m_{pre}$; $s[idx_n] = s_{pre}$;
        end
    end
end
```

After we obtain $\boldsymbol{s}^l$, we can compute the spatial part of $\nabla \boldsymbol{s}^l$ and $\nabla \boldsymbol{w}^l$ & $\nabla \boldsymbol{b}^l$ as Figure 4.4(b) by calling the functions in *cuDNN*. Then, we need to compute $\nabla \boldsymbol{x}^l$ and $\nabla \boldsymbol{\gamma}^l$ & $\nabla \boldsymbol{\beta}^l$ as shown in Figure 4.4(c). Based on Equation 4.3, in order to get $\nabla \boldsymbol{x}^l$, we need to get the sum of $\nabla \boldsymbol{y}^l \hat{\boldsymbol{x}}^l$ and $\nabla \boldsymbol{y}^l$ for each channel first, which are $\nabla \boldsymbol{\gamma}^l$ and $\nabla \boldsymbol{\beta}^l$ as Equation 4.5-4.6. Thus, we first compute the gradients of BN parameters as the intermediate data in BP and PU stages, as given in Algorithm 2. Based on Equation 2.5, the gradients of membrane potentials at the current time step has data dependency on the next time step, therefore $\nabla \boldsymbol{m}^l$ is calculated along time steps from T *to* 1, which is opposed to that in Algorithm 1. The thread model in Algorithm 2 is also based on Figure 4.5.

Algorithm 2: Fused PU Kernel with BN

```
**Data:** $\boldsymbol{x}$, $\boldsymbol{m}$, $\nabla \boldsymbol{s}_{spatial} \in \mathbb{R}^{T\times B\times C\times H\times W}$; $\boldsymbol{\mu}$, $\boldsymbol{\sigma} \in \mathbb{R}^{C}$;
**Result:** $\nabla \boldsymbol{y}_{sum}$, $\nabla \hat{\boldsymbol{y}}_{sum} \in \mathbb{R}^{C\times H\times W}$;
begin
    $\mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$;
    for $b \leftarrow 1$ to $B$ do
        $\nabla m_{next} = 0$;
        for $t \leftarrow T$ to 1 do
            // Eq. (2.4) & (2.6)
            if $th_l < m[idx_n] < th_r$ then
                $\nabla s_{tmp} = \nabla s_{spatial}[idx_n] - \nabla m_{next}(\alpha m[idx_n])$;
            end
            else
                $\nabla s_{tmp} = 0$;
            end
            // Eq. (2.5)
            $\nabla m_{next} = \nabla m_{next}\alpha(m[idx_n] < th_f) + \eta \nabla s_{tmp}$;
            // Eq. (4.4)
            $\hat{x}_{tmp} = \frac{x[idx_n]-\mu_{tmp}}{\sqrt{\sigma_{tmp}^2+\epsilon}}$;
            // Eq. (4.3), (4.5) & (4.6)
            $\nabla y_{sum}[idx_{thread}] += \nabla m_{next}$;
            $\nabla \hat{y}_{sum}[idx_{thread}] += \nabla m_{next}\hat{x}_{tmp}$;
        end
    end
end
```

At last, we can obtain $\nabla \boldsymbol{x}^l$ according to Algorithm 3. In order to avoid extra memory consumption, we do not store the gradients of BN results (i.e., $\nabla \boldsymbol{y}^l$). Therefore, in Algorithm 3, we recompute $\nabla \boldsymbol{m}^l$ again, and go through the time steps from T *to* 1 considering the data dependency in the calculation of $\nabla \boldsymbol{m}$.

After going through all the required kernels in BP and PU stages, we give the overall function in Algorithm 4. The inputs include the Conv results of the current layer (i.e., $\boldsymbol{x}^l$), gradients of Conv results of the next layer (i.e., $\nabla \boldsymbol{x}^{l+1}$), Conv and BN parameters.

Algorithm 3: Fused BP Kernel with BN

```
Data: $\boldsymbol{x}$, $\boldsymbol{m}$, $\nabla\boldsymbol{s}_{spatial} \in \mathbb{R}^{T\times B\times C\times H\times W}$; $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, $\boldsymbol{\gamma}$, $\nabla\boldsymbol{y}_{mean}$, $\nabla\hat{\boldsymbol{y}}_{mean} \in \mathbb{R}^{C}$;
Result: $\nabla\boldsymbol{x} \in \mathbb{R}^{T\times B\times C\times H\times W}$;
begin
    $\mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$; $\gamma_{tmp} = \gamma[j]$;
    $\nabla y_{mean\_tmp} = \nabla y_{mean}[j]$; $\nabla\hat{y}_{mean\_tmp} = \nabla\hat{y}_{mean}[j]$
    for $b \leftarrow 1$ to $B$ do
        for $t \leftarrow 1$ to $T$ do
            Repeat line 7∼19 in Alg. 2 to get $\nabla m_{next}$ and $\hat{x}_{tmp}$;
            // Eq. (4.3)
            $\nabla x[idx_n] = \frac{\gamma_{tmp}}{\sqrt{\sigma^2_{tmp}+\epsilon}}(\nabla m_{next} - \nabla y_{mean\_tmp} - \hat{x}_{tmp}\nabla\hat{y}_{mean\_tmp})$;
        end
    end
end
```

Algorithm 4: PU & BP *CUDA*

```
Data: $\boldsymbol{x}^l$, $\boldsymbol{\mu}^l$, $\boldsymbol{\sigma}^l$, $\boldsymbol{\gamma}^l$, $\boldsymbol{\beta}^l$, $\boldsymbol{w}^{l+1}$, $\nabla\boldsymbol{x}^{l+1}$;
Result: $\nabla\boldsymbol{x}^l$, $\nabla\boldsymbol{\gamma}^l$, $\nabla\boldsymbol{\beta}^l$, $\nabla\boldsymbol{w}^{l+1}$, $\nabla\boldsymbol{b}^{l+1}$;
begin
    // Apply FP to recompute $\boldsymbol{u}^l$ & $\boldsymbol{s}^l$
    $\boldsymbol{m}^l$, $\boldsymbol{s}^l \leftarrow$ Alg. 1 ($\boldsymbol{x}^l$, $\boldsymbol{\mu}^l$, $\boldsymbol{\sigma}^l$, $\boldsymbol{\gamma}^l$, $\boldsymbol{\beta}^l$);
    // Apply PU to get gradients of parameters and intermediate data
    // Eq. (2.7), (2.8), and (2.4)
    $\nabla\boldsymbol{w}^{l+1} = \nabla\boldsymbol{x}^{l+1} * \boldsymbol{s}^l$; $\nabla\boldsymbol{b}^{l+1} = \sum\nabla\boldsymbol{x}^{l+1}$;
    $\nabla\boldsymbol{s}^l_{spatial} = \nabla\boldsymbol{x}^{l+1} * \boldsymbol{w}^{l+1}$;
    $\nabla\boldsymbol{y}^l_{sum}$, $\nabla\hat{\boldsymbol{y}}^l_{sum} \leftarrow$ Alg. 2 ($\boldsymbol{x}^l$, $\boldsymbol{m}^l$, $\nabla\boldsymbol{s}^l_{spatial}$, $\boldsymbol{\mu}^l$, $\boldsymbol{\sigma}^l$);
    // Eq. (4.5) & (4.6)
    $\nabla\boldsymbol{\gamma}^l$, $\nabla\boldsymbol{\beta}^l \leftarrow$ sum of $\nabla\hat{\boldsymbol{y}}^l_{sum}$, $\nabla\boldsymbol{y}^l_{sum}$;
    $\nabla\hat{\boldsymbol{y}}^l_{mean}$, $\nabla\boldsymbol{y}^l_{mean} \leftarrow \nabla\boldsymbol{\gamma}^l/(TBHW)$, $\nabla\boldsymbol{\beta}^l/(TBHW)$;
    // Apply BP to get $\nabla\boldsymbol{x}^l$
    $\nabla\boldsymbol{x}^l \leftarrow$ Alg. 3 ($\boldsymbol{x}^l$, $\boldsymbol{m}^l$, $\nabla\boldsymbol{s}^l_{spatial}$, $\boldsymbol{\mu}^l$, $\boldsymbol{\sigma}^l$, $\boldsymbol{\gamma}^l$, $\nabla\hat{\boldsymbol{y}}^l_{mean}$, $\nabla\boldsymbol{y}^l_{mean}$);
end
```

The outputs contain the gradients of Conv results (i.e., $\nabla \boldsymbol{x}^l$), gradients of Conv and BN parameters.

In order to make our optimization easy-to-deploy, we design Pytorch interfaces[1] which can also support distributed learning.

4.3 Evaluation

4.3.1 Experimental Setup

In this work, we focus on the widely adopted image recognition tasks for evaluation. Our experiments are implemented on two popular datasets: CIFAR10 and ImageNet. The network configurations are detailed in Tab. 4.1. For the CIFAR10 dataset, we design three models with different number of Conv layers; for the imageNet dataset, we adopt an AlexNet-like network and ResNet34 to evaluate our optimization framework. We use a vanilla Pytorch implementation [100] for the BPTT-based SNN training as the baseline which contains BN operations.

4.3.2 Performance Analysis on CIFAR10

We first analyze the improvement of our optimization framework on the CIFAR10 dataset. Figure 4.6(a) shows the comparison without BN. From the results, our optimization framework consumes much less memory for larger models, since the non-optimized functions (e.g., data processing) for small networks consume a large portion of memory. The peak memory consumption under different number of time steps presents little variance, since the portion of saved memory for each layer does not change. For the training latency, our optimization achieves higher speedup as the number of time steps increases.

[1] https://github.com/liangling76/snn_gpu_training_bptt

Dataset	**Model Size**	**Accuracy**	**Network Structure**	T	B	α	η	th_f	(th_l, th_r)
Cifar10	Small	83.17%	I^0 - 128C - 256C - 512C - 1024FC - 512FC - 10FC	10	64	0.25	1.0	0.25	(-0.25, 0.75)
	Medium	88.21%	I^0 - 128C - 128C - 256C - 256C - 512C - 512C - 1024FC - 512FC - 10FC						
	Large	90.17%	I^0 - 128C - 128C - 128C - 256C - 256C - 256C - 512C - 512C - 512C - 1024FC - 512FC - 10FC						
ImageNet	AlexNet	54.94%	I^0 - 64C - 128C - 256C - 256C - 384C - 256C - 256C - 4096FC - 4096FC - 1000FC	6	16				
	ResNet34	60.52%	I^0 - 64C - ⟨ 64B0 - 2×64B1 ⟩ - ⟨ 128B0 - 3×128B1 ⟩ - ⟨ 256B0 - 5×256B1 ⟩ - ⟨ 512B0 - 2×512B1 ⟩ - AP - 1000FC mB0: ⟨ I^l - mC - mC ⟩ + ⟨ I^l - mC ⟩; mB1: ⟨ I^l - mC - mC) + ⟨ I^l ⟩						

Table 4.1: Model configurations. AP-average pooling; mC-a Conv layer that has m output channels; I^0-input; I^l-feature map in layer l.

The reason is that our kernel functions fuse the neural dynamics at all time steps. On the contrary, the vanilla Pytorch updates the neural activities step by step which causes more kernel launching time. For the models with different size, the speedup values are similar, indicating that our optimization can provide a considerable speedup no matter what the network structure is. Overall, for the models without BN, we can achieve maximum 2.13× end-to-end speedup and consume only 0.41× peak memory when compared to the vanilla Pytorch implementation.

After involving the BN layer, our framework achieves less speedup but more memory saving as Figure 4.6(b). The reason is that the BN layer introduces more intermediate data recompute. Overall, for the models with BN, we can achieve maximum 1.94× end-to-end speedup and consume only 0.33× peak memory when compared to the vanilla Pytorch implementation.

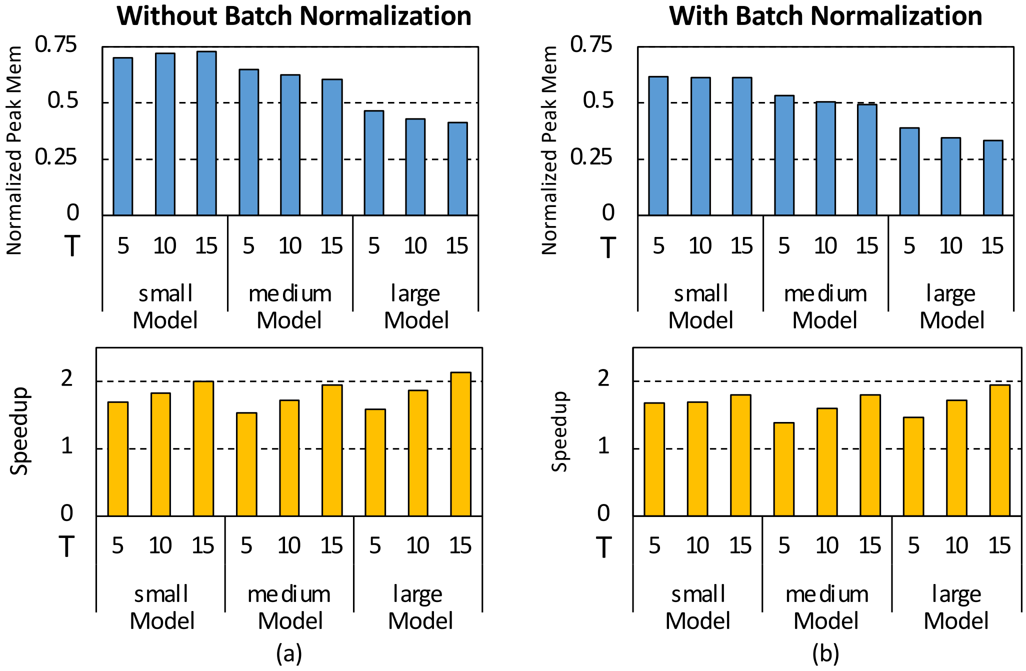


Figure 4.6: Peak memory consumption and performance comparison between our optimization framework and the vanilla Pytorch implementation: (a) without BN; (b) with BN.

We further analyze the breakdown of memory consumption and computation time in Fig 4.7. The vanilla Pytorch spends a higher portion of memory on feature map storage and takes a lower portion of time for the kernel execution. This phenomena demonstrates the efficiency of our framework in saving memory consumption and preventing frequent kernel launching.

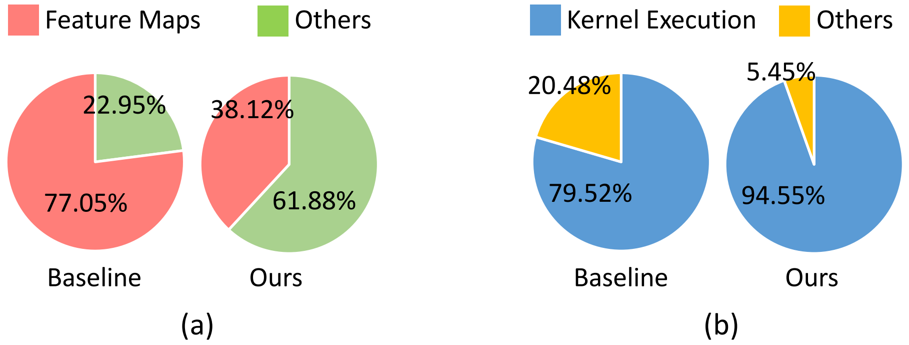


Figure 4.7: Comparison of the breakdown of (a) memory consumption and (b) computation time between the vanilla Pytorch implementation and our optimization framework for the medium-size model with BN and $T = 10$.

4.3.3 Comparison with TorchScript

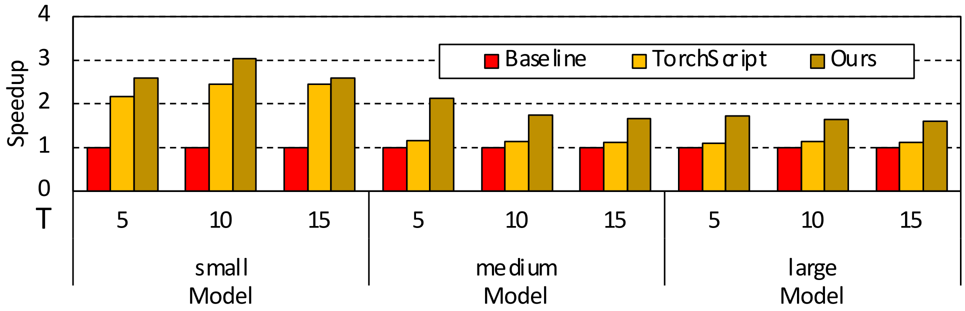


Figure 4.8: Performance comparison in the FP stage between our optimization framework and the Pytorch optimization with TorchScript on Cifar10.

The vanilla Pytorch implementation for the FP stage can be easily optimized through TorchScript [101]. We compare our framework with TorchScript in Figure 4.8. From the results, both frameworks achieve high speedup for small models, since a lower portion of time is spent on Conv/FC layers. Our framework and Torchscript fuse operations in one layer along the temporal and spatial directions, respectively. For larger models, our framework achieves a higher speedup that indicates fusing operations along the temporal direction is more efficient.

4.3.4 Performance Analysis on ImageNet

Finally, we analyze our optimization framework on the ImageNet dataset with distributed learning. The comparison to the vanilla Pyotrch implementation is shown in Figure 4.9. For the peak memory consumption per GPU, we find that the vanilla Pytorch implementation needs much more storage in distributed learning compared to the single-GPU training. The reason is that the vanilla Pytorch implementation induces extra memory overhead for some intermediate data such as $\hat{\boldsymbol{x}}_t^l$. In our framework, the additional storage consumption only comes from the calculation of the average parameter gradients between GPUs. Compared to the vanilla Pytorch impelementation, our optimization framework only consumes 0.68× and 0.38× peak memory for ResNet34 and AlexNet, respectively.

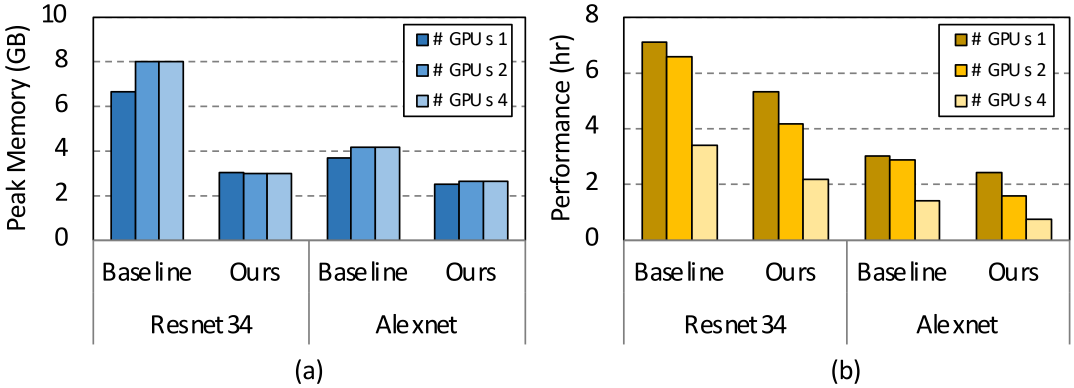


Figure 4.9: Comparison between our optimization framework and the vanilla Pytorch implementation for distributed learning on the ImageNet dataset: (a) peak memory consumption per GPU; (b) training latency for one epoch.

Compared to the single-GPU training, both frameworks achieve insignificant speedup when using two GPUs. The reason is that, the distributed training needs additional communication between GPUs, such as parameter synchronization. When further scaling up to more GPUs, the latency is dramatically reduced due to the enhanced computing power but the similar communication burden for each GPU compared to the case with two GPUs. From the simulation results, our optimization framework can achieve maximum 1.58× and 1.81× end-to-end speedup for ResNet34 and Alexnet, respectively, when compared to the vanilla Pytorch implementation.

4.4 Conclusion

This Chapter describes a framework that accelerates BPTT-based SNN training on GPU. The framework first optimizes the dataflow by only storing the Conv results in the FP stage to reduce the memory consumption. Then, several customized kernel functions are designed to alleviate frequent kernel launching. Also, the Pytorch interfaces are provided to make the framework easy-to-deploy.

Chapter 5

Exploring Adversarial Attack in Spiking Neural Networks with Spike-Compatible Gradient

The previous two Chapters describe how to make SNN training efficient by designing an accelerator and optimizing GPU kernels. In this Chapter, a new topic will be discussed, which is the security of SNNs. As SNNs' accuracy on general tasks becomes higher, SNN security becomes important while lacking in-depth investigation. To this end, this Chapter explores attacking an SNN model through adversarial examples, which is well studied in ANNs.

5.1 Preliminaries and Challenges

In this Section, we first introduce the adversarial attack in ANNs. Then, the challenges of attack SNNs via adversarial examples are illustrated.

5.1.1 Gradient-based Adversarial Attack

We take the gradient-based adversarial attack in ANNs as an illustrative example. The neural network is actually a map from inputs to outputs, i.e., $y = f(x)$, where x and y denote inputs and outputs, respectively, and $f : R^m \to R^n$ is the map function. Usually, the inputs are static images in convolutional neural networks. In adversarial attack, the attacker attempts to manipulate the victim model to produce incorrect outputs by adding imperceptible perturbations δ in the input images. We define $x' = x + \delta$ as an adversarial example. The perturbation is constrained by $\|\delta\|_p = \|x' - x\|_p \leq \epsilon$, where $\|\cdot\|_p$ denotes the p-norm and ϵ reflects the maximum tolerable perturbation.

Generally, the adversarial attack can be categorized into untargeted attack and targeted attack according to the different attack goals. Untargeted attack fools the model to classify the adversarial example into any other classes except for the original correct one, which can be illustrated as $f(x + \delta) \neq f(x)$. In contrast, for targeted attack, the adversarial example must be classified in to a specified class, i.e., $f(x+\delta) = y_{target}$. With these preliminary knowledge, the adversarial attack can be formulated as an optimization problem as below to search the smallest perturbation:

$$\begin{cases} \arg\min\limits_{\delta} \|\delta\|_p, \ \ s.t. \ f(x + \delta) \neq f(x), \quad \text{if untargeted} \\ \arg\min\limits_{\delta} \|\delta\|_p, \ \ s.t. \ f(x + \delta) = y_{target}, \quad \text{if targeted} \end{cases} . \tag{5.1}$$

There are several widely-adopted adversarial attack algorithms to find an approximated solution. Here we introduce two of them: the fast gradient sign method (FGSM) [38] and the basic iterative method (BIM) [39].

FGSM. The main idea of FGSM is to generate the adversarial examples based on the gradient information of the input. Specifically, it calculates the gradient map of an input image, and then adds or subtracts the *sign* of this input gradient map in the original image with multiplying a small scaling factor. The generation of adversarial examples can be formulated as

$$\begin{cases} x' = x + \eta \cdot sign(\nabla_x L(\theta, x, y_{orginal})), \text{ if untargeted} \\ x' = x - \eta \cdot sign(\nabla_x L(\theta, x, y_{target})), \text{ if targeted} \end{cases} \tag{5.2}$$

where L and θ denote the loss function and parameters of the victim model. η is used to control the magnitude of the perturbation. In untargeted attack, the adversarial example will drive the output away from the original correct class, which results from the gradient ascent-based input modification; while in targeted attack, the output under the adversarial example goes towards the targeted class, owing to the gradient descent-based input modification.

BIM. BIM algorithm is actually the iterative version of the above FGSM, which updates the adversarial examples in an iterative manner until the attack succeeds. The generation of adversarial examples in BIM is governed by

$$\begin{cases} x'_{k+1} = x'_k + \eta \cdot sign(\nabla_{x'_k} L(\theta, x'_k, y_{orginal})), \text{ if untargeted} \\ x'_{k+1} = x'_k - \eta \cdot sign(\nabla_{x'_k} L(\theta, x'_k, y_{target})), \text{ if targeted} \end{cases} \tag{5.3}$$

where k is the iteration index. Specifically, x'_k equals the original input when $k = 0$.

In ANNs, several advanced attack methods can be potentially extended beyond BIM based algorithm by optimizing the perturbation bound [40, 41, 42, 43] or avoiding the gradient calculation [102, 103, 104, 105]. In this work, we aim at the preliminary exploration of an effective gradient-based SNN attack, thus adopting the most classic BIM algorithm in our design.

5.1.2 Challenges in SNN Attack

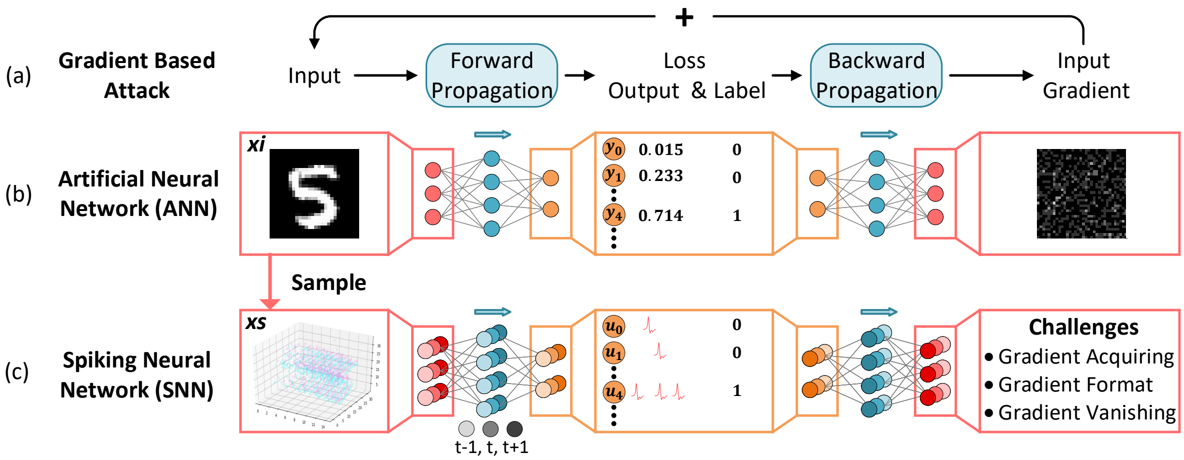


Figure 5.1: Illustration of gradient-based adversarial attack: (a) overall flow including forward pass, backward pass, and input update; (b) adversarial attack in ANNs; (c) adversarial attack in SNNs and its challenges. xi and xs represent an input in image and spike formats, respectively.

Even though the attack methodology can be independent of how the model is trained (e.g., gradient-free unsupervised learning [106] or spatial-gradient-based supervised learning [70]) and it is not necessary to compute gradients when finding adversarial examples (e.g., using trial-and-error methods [107, 71]), we take SNN models trained by BPTT with high recognition accuracy for example and focus on the spatiotemporal-gradient-based attack due to the potential for high attack success rate. Therefore, all our following discussions about the challenges are restricted in this context. Figure 5.1(a) briefly illustrates the work flow of adversarial attack based on gradients. There are three stages: forward pass to obtain the model prediction, backward pass to calculate the input gradient, and input update to generate the adversarial example. This flow is straightforward to implement in ANNs, as shown in Figure 5.1(b). However, the case becomes complicated in the SNN scenario, where the processing is based on binary spikes with temporal dynamics rather than continuous activations with immediate response. According to Figure 5.1(c), we attempt to identify the challenges in SNN attack to distinguish from the ANN attack and compare our solution with prior studies.

Challenge1: Acquiring Spatiotemporal Gradients.

For SNNs, it is difficult to acquire the spatiotemporal gradients using conventional SNN learning algorithms for the generation of adversarial examples with both spatial and temporal components. For example, the unsupervised learning rules such as spike timing dependent plasticity (STDP) [13] update synapses according to the activities of local neurons, which cannot help calculate the input gradients. The ANN-to-SNN-conversion learning methods [44] simply convert an SNN learning problem into an ANN one with only spatial information, leading to the incapability in capturing temporal input gradients. Recently, the backpropagation through time (BPTT) based learning algorithm [20, 21, 22, 6, 3, 24] is broadly studied. This emerging supervised learning promises accurate SNN attack via the direct acquisition of input gradients in both spatial and temporal dimensions, which is adopted by us.

Challenge2: Incompatible Format between Gradients and Inputs.

The input gradients are in continuous values, while the SNN inputs are in binary spikes (see the left of Figure 5.1(c), each point represents a spike event, i.e., “1”; otherwise it is “0”). This data format incompatibility impedes the generation of spike-based adversarial examples if we consider the conventional gradient accumulation. In this work, we propose a gradient-to-spike (G2S) converter to convert continuous gradients to spike-compatible ternary gradients. This design exploits probabilistic sampling, sign extraction, and overflow-aware transformation, which can simultaneously maintain the spike format and control the perturbation magnitude.

Gradient Vanishing Problem.

The firing function in the LIF model in Equation (2.1) is actually a step function that is non-differentiable. To address this issue, an approximation function is introduced to simulate the derivative of the firing activity [20]. However, this approximation brings abundant zero gradients outside the gradient window (to be shown latter), leading to severe gradient vanishing during backpropagation. We find that the input gradient map can be all-zero sometimes, which interrupts the gradient-based update of adversarial examples. To this end, we propose a restricted spike flipper (RSF) to construct ternary gradients that can randomly flip the binary inputs in the case of all-zero gradients. We use a baseline sampling factor to bound the overall turnover rate, making the perturbation magnitude controllable. Further more, we propose threshold tuning to deal with the gradient vanish problem which cannot be handled by RSF.

5.1.3 Comparison with Prior Work on SNN Attack

The study on SNN attack is still in its infant stage. In this subsection, we summarize existing approaches and clarify our differences compared with them.

Attack Method	**Data Source**	**SpatialTemporal Gradient**	**Computational Complexity**	**Attack Effectiveness**
Trial-and-Error [107]	Image	✗	$Iter \times N \times 2C_{FP}$	Low
Trial-and-Error [71]	Spike	✗	$Iter \times N \times C_{FP}$	Low
Model Conversion [70]	Image	✗	$Iter \times (C_{FP} + C_{BP})$	Low
This Work	Spike/Image	✓	$Iter \times (C_{FP} + C_{BP})$	High

Table 5.1: Comparison with prior work on SNN attack.

Trial-and-Error Input Perturbation.

Such attack algorithms perturb inputs in a trial-and-error manner by monitoring the variation of outputs. For example, A. Marchisio et al. [107] modify the original image inputs before spike sampling. They first select a block of pixels in the images, and then add a positive or negative unit perturbation onto each pixel. During this process, they always monitor the output change to determine the perturbation until the attack succeeds or the perturbation exceeds a threshold. However, this image-based perturbation is not suitable for the data sources with only spike events [82, 83]. In contrast, A. Bagheri et al. [71] directly perturb the spike inputs rather than the original image inputs. The main idea is to flip the input spikes and also monitor the outputs.

SNN/ANN Model Conversion.

S. Sharmin et al. [70] convert the SNN attack problem into an ANN one. They first build an ANN substitute model that has the same network structure and parameters copied from the trained SNN model. The gradient-based adversarial attack is then conducted on the built ANN counterpart to generate the adversarial examples.

These existing works suffer from several drawbacks that would eventually degrade the attack effectiveness. For the trial-and-error input perturbation methods, the computational complexity is quite high due to the large search space without the guidance of supervised gradients. Specifically, each selected element of the inputs needs to run the forward pass once (for spike perturbation) or twice (for image perturbation) to monitor the outputs. The total computational complexity is $Iter \times N \times C_{FP}$, where $Iter$ is the number of attack iterations, N represents the size of search space, and C_{FP} is the computational cost of each forward pass. This complexity is much higher than the normal one, i.e., $Iter \times (C_{FP} + C_{BP})$, due to the large N. Because it is difficult to find the optimal perturbation in such a huge space, the attack effectiveness cannot be satisfactory given a limited search time in reality. Regarding the SNN/ANN model conversion method, an extra model transformation is needed and the temporal gradient information is lost during the ANN pretraining. Using a different model to find gradients and the missing of temporal components will compromise the attack effectiveness in the end. Moreover, this method is not applicable to the spiking data sources without the help of extra signal conversion.

Compared with the above works we calculate the gradients in both spatial and temporal dimensions without extra model conversion, which matches the natural SNN behaviors. Then, the proposed G2S and RSF enable the generation of spiking adversarial examples based on the continuous gradients even if when meeting the gradient vanishing. This direct generation of spiking adversarial examples makes our methodology suitable for the spiking data sources. For the SNN models using image-based data sources, our solution is also applicable with a simple temporal aggregation of spatiotemporal gradients. In summary, Table 5.1.3 shows the differences between our work and prior work. We use effectiveness to assess an attack method. Usually, an effective attack method can achieve high attack success rate with relative low complexity and better compatibility of input data formats.

Please note that we focus on the white-box attack in this paper. Specifically, in the white-box attack scenario, the adversary knows the network structure and model parameters (e.g., weights, u_{th}, etc.) of the victim model. The reason of this scenario selection lies in that the white-box attack is the fundamental step to understand adversarial attack. Furthermore, the methodology built for the white-box attack can be easily transferred to the black-box attack in the future.

5.2 Adversarial Attack Against SNNs

In this section, we first introduce the input data format briefly, and then explain the flow, approach, and algorithm of our attack methodology in detail.

5.2.1 Input Data Format

It is natural for an SNN model to handle spike signals. Therefore, considering the datasets containing spike events, such as N-MNIST [82] and CIFAR10-DVS [83], is the first choice. In this case, the input is originally in a spatiotemporal pattern with a binary value for each element (0-nothing; 1-spike). The attacker can flip the state of selected elements, while the binary format must be maintained. The image datasets are also widely used in the SNN field by converting them into the spiking version. There are different ways to perform the data conversion, such as rate coding [3, 6, 90] and latency coding [108, 109, 110]. In this work, we adopt the former scheme based on Bernoulli sampling that converts the pixel intensity to a spike train, where the spike rate is proportional to the intensity value. In this case, the attacker can modify the intensity value of selected pixels by adding the continuous perturbation. Figure 5.2 illustrates the adversarial examples in these two cases.

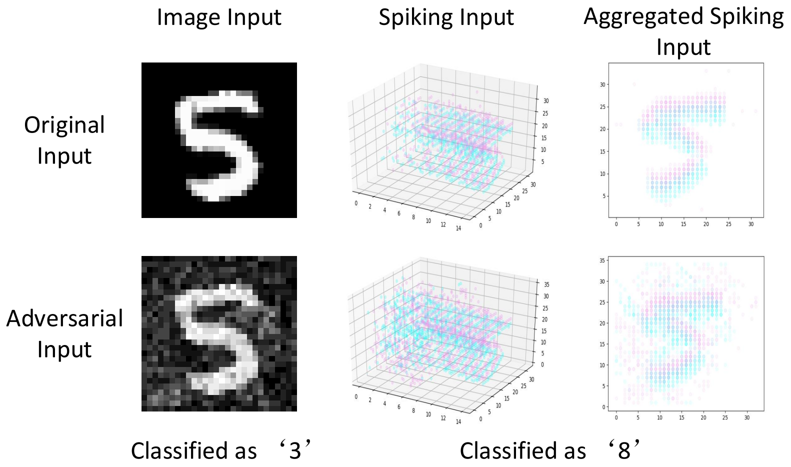


Figure 5.2: The data format of original inputs and adversarial examples. The red and blue colors denote two spike channels induced by dynamic vision sensors [84, 82].

5.2.2 Attack Flow Overview

The overview of the proposed adversarial attack against SNNs is illustrated in Figure 5.3. The basic flow adopts the BIM method given in Equation (5.3).

The perturbation for spikes can only flip the binary states of selected input elements rather than add continuous values. Therefore, to generate spiking adversarial examples, the search of candidate elements is more important than the perturbation magnitude. FGSM cannot do this since it only explores the perturbation magnitude, while BIM realizes this by searching new candidate elements in different attack iterations. Next, we describe the specific flow for spiking inputs and image inputs individually.

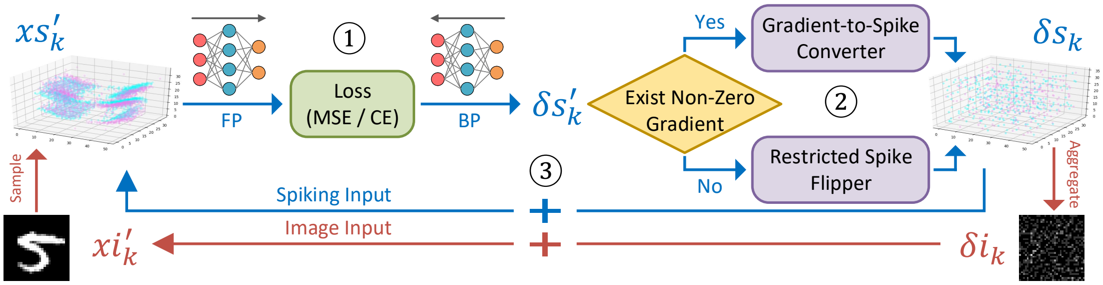


Figure 5.3: Overview of the adversarial attack flow for SNNs with spiking or image inputs. The flow consists of: ① calculating continuous spatiotemporal input gradients via BPTT; ② generating spike-compatible input gradients; ③ updating adversarial examples. For image-based inputs, an additional aggregation of the input gradients along the temporal dimension is needed.

Spiking Inputs

The blue arrows in Figure 5.3 illustrate this case. The generation of spiking adversarial examples relies on three steps as follows. In step ①, the continuous gradients are calculated in the FP and BP stages by

$$\begin{cases} \delta s'_k = \nabla_{xs'_k} L(\theta, xs_k, y_{original}), \text{ if untargeted} \\ \delta s'_k = -\nabla_{xs'_k} L(\theta, xs_k, y_{target}), \text{ if targeted} \end{cases} \tag{5.4}$$

where $\delta s'_k$ represents the input gradient at the k-th iteration. Since all elements in $\delta s'_k$ are continuous values, they cannot be directly accumulated onto the spiking inputs xs_k. Therefore, in the step ②, we propose G2S to convert the continuous gradient to a ternary one compatible with the spike input, which can simultaneously maintain the input data format and control the perturbation magnitude. When the input gradient vanishes (i.e., all elements in $\delta s'_k$ are zero), we propose RSF to construct a ternary gradient that can randomly flip the input spikes with a controllable turnover rate. At last, step ③ accumulates the ternary gradients onto the spiking input.

Image Inputs

Sometimes, the benchmarking models convert image datasets to spike inputs via Bernoulli sampling. In this case, one more step is needed to generate image-style adversarial examples, which is shown by the red arrows in Figure 5.3. After the above step ②, the ternary gradient map should be aggregated in the temporal dimension according to $\delta i_k = \frac{1}{T}\sum_{t=1}^{T} \delta s_k^t$. In each update iteration, the intensity value of xi_k will be clipped within $[0,\ 1]$.

5.2.3 Gradient-to-Spike (G2S) Converter

There are two goals in the design of G2S converter in each attack iteration: (1) the final gradients should be compatible with the spiking inputs, i.e., keeping the spike format unchanged after the gradient accumulation; (2) the perturbation magnitude should be imperceptible, i.e., limiting the number of non-zero gradients. To this end, we design three steps: probabilistic sampling, sign extraction, and overflow-aware transformation, which are illustrated in Figure 5.4.

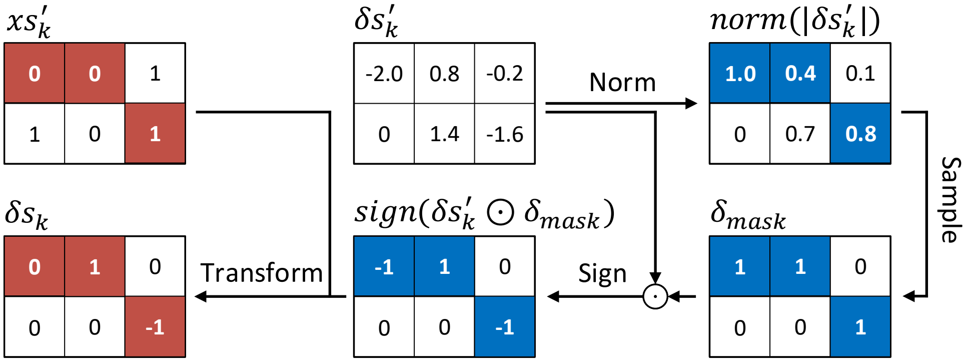


Figure 5.4: Illustration of gradient-to-spike (G2S) converter with probabilistic sampling reducing the number of modified points, sign extraction ternarizing the continuous gradients for spike compatibility, and overflow-aware transformation clipping the data range in adversarial examples.

Probabilistic Sampling

The absolute value of the input gradient $|\delta s_k'|$ obtained by Equation (5.4) is first normalized to lie in the range of $[0,\ 1]$. Then, the normalized gradient map $norm(|\delta s_k'|)$ is sampled to produce a binary mask, in which the 1s indicate the locations where gradients can pass through. The probabilistic sampling for each gradient element obeys

$$\begin{cases} P(\delta_{mask} = 1) = norm(|\delta s_k'|) \\ P(\delta_{mask} = 0) = 1 - norm(|\delta s_k'|) \end{cases} . \tag{5.5}$$

By multiplying the resulting mask with the original gradient map, the number of non-zero elements can be reduced significantly. To evidence this conclusion, we run the attack against the SNN model with a network structure to be provided in Table 5.4.1 over 500 spiking inputs from N-MNIST, and the results are presented in Figure 5.5. Given MSE loss and untarget attack scenario, the number of non-zero elements in $\delta s_k'$ could reach 2^{10}. After using the probabilistic sampling, the number of non-zero elements in $\delta s_k' \odot \delta_{mask}$ can be greatly decreased, masking out $> 96\%$ percentage.

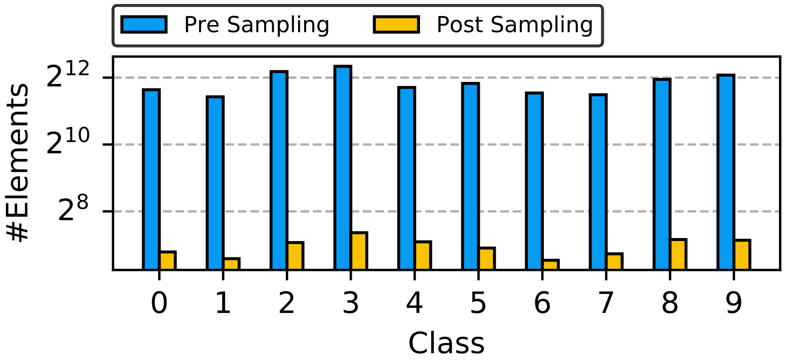


Figure 5.5: The number of elements with non-zero input gradients before and after the probabilistic sampling. After the probabilistic sampling step, the number of selected non-zero input elements for modification is reduced a lot for each class.

Sign Extraction

Now, we explain how to generate a ternary gradient map where each element is in $\{-1,\ 0,\ 1\}$ to match the spike inputs. This step is simply based on a sign extraction:

$$\delta s''_k = sign(\delta s'_k \odot \delta_{mask}) \tag{5.6}$$

where we define $sign(x) = 1$ if $x > 0$, $sign(x) = 0$ if $x = 0$, and $sign(x) = -1$ otherwise.

Overflow-aware Transformation.

Although the above $\delta s''_k$ is able to be ternary, it cannot ensure that the final adversarial example generated by input gradient accumulation is still limited in $\{0,\ 1\}$. For example, an original "0" element in xs_k with a "-1" gradient or an original "1" element with a "1" gradient will yield a "-1" or "2" input that is out of $\{0,\ 1\}$. This overflow breaks the data format of binary spikes. To address this issue, we propose an overflow-aware gradient transformation to constrain the range of the final adversarial example, which is illustrated in Table 5.2.3.

After introducing the above three steps, now the function of G2S converter can be

	Before Transformation		After Transformation	
xs'_k	$\delta s''_k$	$xs_k + \delta s''_k$	δs_k	$xs'_k + \delta s_k$
0/1	0	0/1	0	0/1
0	1	1	1	1
1	1	2	0	1
0	-1	-1	0	0
1	-1	0	-1	0

Table 5.2: Overflow-aware gradient transformation.

briefly summarized as below:

$$\delta s_k = transform[sign(\delta s'_k \odot \delta_{mask}),\ xs'_k] \tag{5.7}$$

where $transform(\cdot)$ denotes the overflow-aware transformation. The G2S converter is able to simultaneously keep the spike compatibility and control the perturbation magnitude.

5.2.4 Restricted Spike Flipper (RSF)

Dataset	N-MNIST	CIFAR10-DVS	MNIST	CIFAR10
#grad.-vanish. inputs (MSE)	130	41	436	103
#grad.-vanish. inputs (CE)	256	32	471	105

Table 5.3: Number of inputs with all-zero gradients at the first attack iteration. We test the untargeted attack with over 500 inputs for each dataset.

Table 5.2.4 identifies the gradient vanishing issue in SNNs, which is quite severe. Based on the previous study [20], the hyper-parameter th_l and th_r in Equation (2.6) has an influential impact on the gradient approximation of the fire function. Generally, a too small $th_r - th_l$ would prevent gradients from passing through the neurons in the backward pass, i.e., aggravating the gradient vanishing problem. However, a too large $th_r - th_l$ cannot precisely approximate the gradient of the firing function that should be a delta function rather than a wide pulse. Therefore, the gradient vanishing problem cannot be fully resolved by simply increasing $th_r - th_l$. Therefore, we empirically select proper th_l and th_r values (see Table 5.4.1) and propose RSF to address the gradient vanishing problem. Specifically, we design two steps for RSF: element selection and gradient construction, which are illustrated in Figure 5.6.

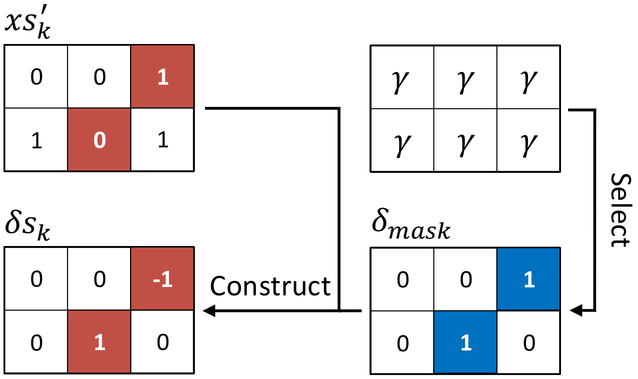


Figure 5.6: Illustration of restricted spike flipper (RSF) with element selection picking candidate elements through probabilistic sampling and gradient construction creating spike-compatible gradients through spike flipping.

Element Selection

This step is to select the elements to flip the spike event. We provide a gradient initialization that sets all elements to γ as the example provided in Figure 5.6. γ is a factor within the range of [0, 1], which controls the number of non-zero gradients after RSF. Now the probabilistic sampling in Equation (5.5) is still applicable to generate the mask δ_{mask}.

Gradient Construction.

To maintain the spike format of adversarial examples, we just flip the state of spiking inputs in the selected region. Table 5.2.4 illustrates the construction of ternary gradients that are able to flip the spiking inputs.

		After Construction	
xs'_k	δ_{mask}	δs_k	$xs'_k + \delta s_k$
0/1	0	0	0/1
0	1	1	1
1	1	-1	0

Table 5.4: Gradient construction to flip spiking inputs.

With the above two steps, the spiking inputs can be flipped randomly with a good control of the turnover rate. The overall function of RSF can be expressed as

$$\delta s_k = construct(\delta_{mask},\ xs'_k). \tag{5.8}$$

5.2.5 Overall Attack Algorithm

Based on the explanations of G2S converter and RSF, Algorithm 5 provides the overall attack algorithm corresponding to the attack flow illustrated in Figure 5.3. There are several hyper-parameters in our algorithm, such as the maximum attack iteration number ($Iter$), the norm format (p) to quantify the perturbation magnitude, the perturbation magnitude upper bound (ϵ), the gradient scaling rate (η), and the sampling factor (γ) in RSF. Notice that we use the average perturbation per point as the metric to evaluate the perturbation magnitude for adversarial example with N pixel points, i.e., $\frac{1}{N}\|x'_{k+1} - x'_0\|_p$.

5.3 Trap Effect in SNN Attack

This Chapter considers two design knobs that affect the SNN attack effectiveness: the loss function during training and the firing threshold of the penultimate layer during attack.

Algorithm 5: The overall SNN attack algorithm.

```
Input: $x$, $Iter$, $p$, $\epsilon$, $\eta$, $\gamma$;
if image input then $xi_0 = x$; end
else $xs'_0 = x$; end
for $k = 1$ to $Iter$ do
    if image input then
        $xs'_k \leftarrow$ Bernoulli sampling on $xi'_k$;
    end
    Get $\delta s'_k$ through Equation (5.4);
    if gradient vanishing occurs in $\delta s'_k$ then
        // RSF
        $\delta_{mask} \leftarrow$ Probabilistic sampling on $\gamma$;
        $\delta s_k = construct(\delta_{mask},\ xs'_k)$;
    end
    else
        // G2S converter
        $\delta_{mask} \leftarrow$ Probabilistic sampling on $norm(|\delta s'_k|)$;
        $\delta s_k = transform[sign(\delta s'_k \odot \delta_{mask}),\ xs'_k]$;
    end
    if image input then
        $\delta i_k \leftarrow \frac{1}{T}\sum_{t=1}^{T} \delta s_k^t$; // Temporal aggregation
        $xi'_{k+1} = xi'_k + \delta i_k$;
        if $\frac{1}{N}\|xi'_{k+1} - xi'_0\|_p \geqslant \epsilon$ then
            break; // Attack failed
        end
        if attack succeeds then
            return $xi'_{k+1}$; // Attack successful
        end
    end
    else
        $xs'_{k+1} = xs'_k + \delta s_k$;
        if $\frac{1}{N}\|xs'_{k+1} - xs'_0\|_p \geqslant \epsilon$ then
            break; // Attack failed
        end
        if attack succeeds then
            return $xs'_{k+1}$; // Attack successful
        end
    end
end
```

5.3.1 MSE and CE Loss Functions

We compare two widely used loss functions, mean square error (MSE) loss and cross entropy (CE) loss. We observe that the gradient vanishing occurs more often when the model is trained by CE loss. It seems that there is a "trap" region in this case. Specifically, the output neurons cannot change the response any more no matter how RSF modifies the input. As shown in Figure 5.7(a), when we use CE loss during training, the gradient is usually vanished between the decision boundaries (i.e., the shaded area); while this phenomenon seldom happens if MSE loss is used.

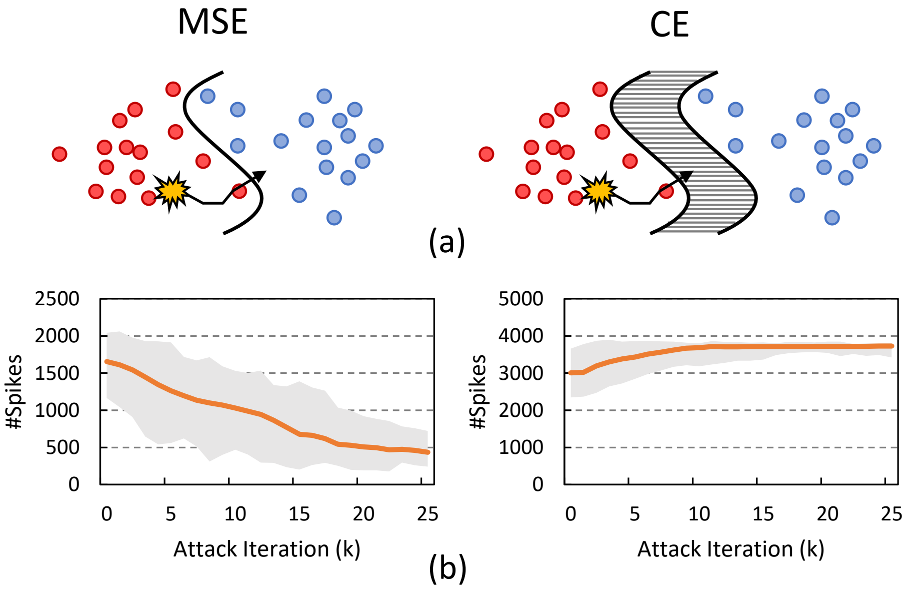


Figure 5.7: Loss function analysis: (a) decision boundary comparison; (b) the number of output spikes in the penultimate layer at different attack iterations. The shaded area in (a) represents a "trap" area that receives zero gradient; the larger number of spikes in the penultimate layer under CE probably introduces the "trap" effect.

For a deeper understanding, we examine the output pattern of the penultimate layer (during untargeted attack) since it directly interacts with the output layer, as depicted in Figure 5.7(b). Here the network structure will be provided in Table 5.4.1 and the 500 test inputs are randomly selected from the N-MNIST dataset. When the training loss is MSE, the number of output spikes in the penultimate layer gradually decreases as the attack process evolves. On the contrary, the spike number first increases and then stays unchanged for the CE trained model. Based on this observation, one possible hypothesis is that more output spikes in the penultimate layer might increase the distance between decision boundaries, thus introducing the mentioned “trap” region with gradient vanishing.

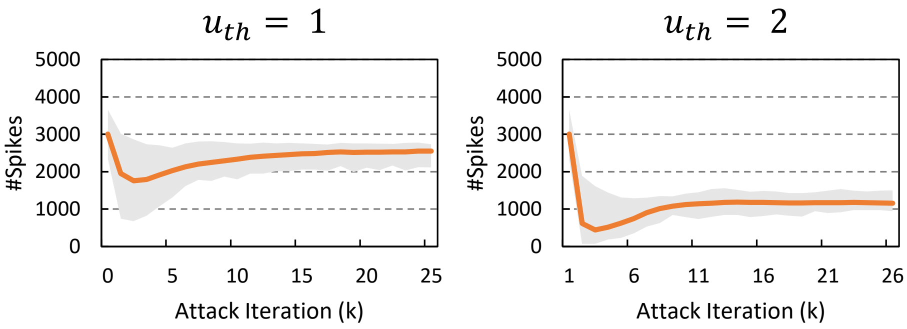


Figure 5.8: The number of output spikes in the penultimate layer with different firing threshold in that layer. The increase of firing threshold in the penultimate layer is able to reduce the number of spikes.

5.3.2 Firing Threshold of the Penultimate Layer

As introduced in the above subsection, the models trained by CE loss are prone to output more spikes in the penultimate layer, leading to the “trap” region that makes the attack difficult. To address this issue, we increase the firing threshold of the penultimate layer during attack to reduce the number of spikes there. Notice that we only modify the firing threshold in the FP stage during the generation of adversarial examples. With the threshold tuning, we present the number of spikes again in Figure 5.8, where the CE loss is used and other settings are the same with those in Figure 5.7(b). Compared to the original threshold setting ($th_f = 0.3$) in the previous experiments, the number of output spikes in the penultimate layer can be decreased significantly on average. Latter experiments in Section 5.4.4 will evidence that this tuning of firing threshold is able to improve the adversarial attack effectiveness.

5.4 Evaluation

5.4.1 Experiment Setup

We design our experiments on both spiking and image datasets. The spiking datasets include N-MNIST [82] and CIFAR10-DVS [83] which are captured by dynamic vision sensors [84]; while the image datasets include MNIST [79] and CIFAR10 [80]. For these two kinds of dataset, we use different network structure, as listed in Table 5.4.1. For each dataset, the detailed hyper-parameter setting during training and the trained accuracy are shown in Table 5.4.1. For each model, we train it for 50 epochs, and the learning rate decays by 0.1 at epoch 35. The default loss function is MSE. Since we focus on the attack methodology in this work, we do not use the optimization techniques such as input encoding layer, neuron normalization, and voting-based classification [6].

Dataset	Network Structure
Spike	Input-128C3-128C3-AP2-384C3-384C3-AP2-1024FC-512FC-10FC
Image	Input-128C3-256C3-AP2-512C3-AP2-1024C3-512C3-1024FC-512FC-10FC
Gesture-DVS	Input-64C3-128C3-AP2-128C3-AP2-256FC-11FC

Table 5.5: Network structure on different datasets. “C”, “AP”, and “FC” denote convolutional layer, average pooling layer, and fully-connected layer, respectively.

Datasets	Gesture-DVS	N-MNIST	CIFAR10-DVS	MNIST	CIFAR10
Input Size	$32 \times 32 \times 2$	$34 \times 34 \times 2$	$42 \times 42 \times 2$	$28 \times 28 \times 1$	$32 \times 32 \times 3$
th_f	0.3	0.3	0.3	0.3	0.3
η	0.3	0.3	0.3	0.25	0.25
$th_r - th_l$	0.5	0.5	0.5	1	1
T	60	15	10	15	15
Time Bin	1ms	5ms	5ms	-	-
Acc (MSE)	91.32%	99.49%	64.60%	99.27%	76.37%
Acc (CE)	-	99.42%	64.50%	99.52%	77.27%

Table 5.6: Hyper-parameter settings and model accuracy during training. $(th_r + th_l = 2 \cdot th_f)$

We set the maximum iteration number of adversarial attack, i.e., $Iter$ in Algorithm 5, to 25. We randomly select 50 inputs in each of the 10 classes for untargeted attack and 10 inputs in each class for targeted attack. In targeted attack, we set the target to all classes except the ground-truth one. We use attack success rate and average perturbation per point (i.e., $\|\delta\|_p$) as two metrics to evaluate the attack effectiveness. Specifically, the attack success rate is calculated in the same way as the prior work do [111, 38]: for untargeted attack, it is the percentage of the cases that adversarial examples fool the model to output a different label from the ground-truth one; for targeted attack, it is the percentage of the cases that adversarial examples manipulate the model to output the target label. Noted, during the calculation of attack success rate, we only consider the original images that can be correctly classified to eliminate the impact of intrinsic model prediction errors. The reason that we use the same perturbation metric of point-to-point distance for both image-based and spike-based data sources is to simplify the comparison. In the perturbation calculation, we adopt L2 norm, i.e., $p = 2$. For image-based datasets, we normalize each input value into $[0, 1]$.

5.4.2 Influence of G2S Converter

We first validate the effectiveness of G2S converter. Among the three steps in G2S converter (i.e., probabilistic sampling, sign extraction, and overflow-aware transformation) as introduced in Section 5.2.3, the last two are needed in addressing the spike compatibility while the first one is used to control the perturbation amplitude. Therefore, we examine how does the probabilistic sampling affects the attack effectiveness. Nnote that we do not use RSF to solve the gradient vanishing here.

Figure 5.9 presents the comparison of attack results over four datasets with or without the probabilistic sampling. In this subsection we only estimate the attack success/failure

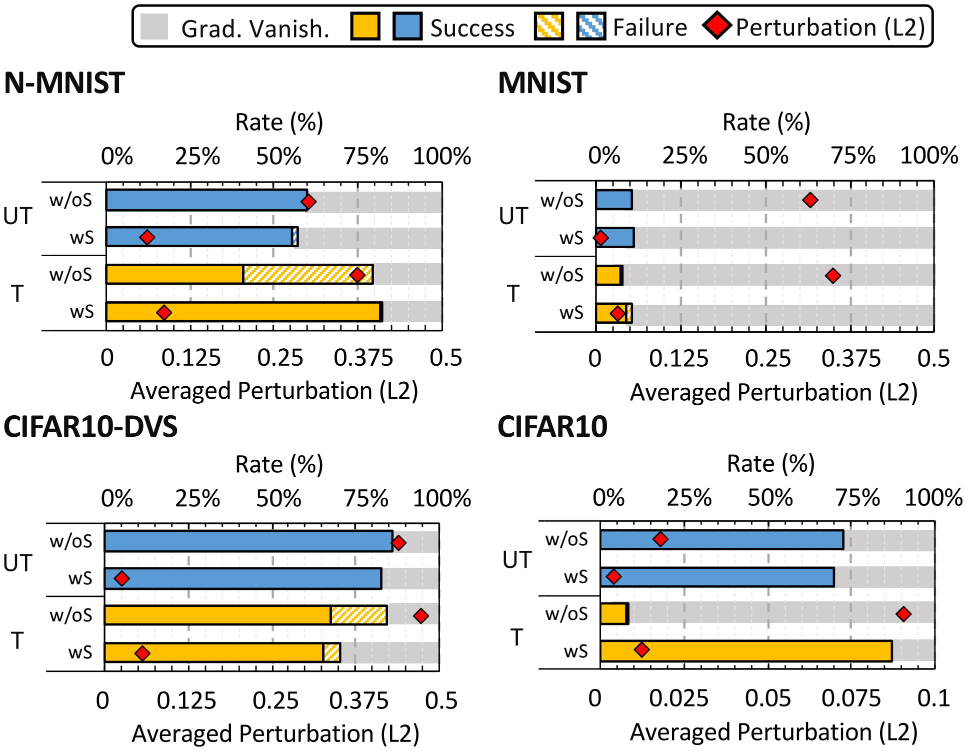


Figure 5.9: Comparison of attack success rate and average perturbation over different datasets with and without probabilistic sampling in G2S converter. “T”, “UT”, “w/oS”, and “wS” refer to targeted attack, untargeted attack, G2S without probabilistic sampling, and G2S with probabilistic sampling, respectively.

rate for the input samples that do not encounter the gradient vanishing problem during attack. Thus, three parts add up to 100%, i.e., the success rate, the failure rate, and the percentage of the input samples that encounter the gradient vanishing problem.

We provide the following observations. First, the required perturbation amplitude of targeted attack is higher than that of untargeted attack, and the success rate of targeted attack is usually lower than that of untargeted attack. These results reflect the difficulty of targeted attack that needs to move the output to an expected class accurately. Second, the probabilistic sampling can significantly reduce the perturbation amplitude in all cases because it removes many small gradients. Third, the probabilistic sampling can maintain the attack success rate in most cases under targeted attack. Specifically, on N-MNIST and CIFAR10 datasets, the probabilistic sampling can improve the targeted attack success rate a lot (e.g., >80% on CIFAR10). Although the targeted attack success rate is slightly lowered after applying the probabilistic sampling on CIFAR10-DVS, it is not the mainstream and might be caused by the restriction on the number of attack iterations. With the probabilistic sampling, the attack failure rate could be reduced to almost zero if the gradient does not vanish.

5.4.3 Influence of RSF

Then, we validate the effectiveness of RSF. In RSF, the hyper-parameter γ controls the number of selected elements, thus affecting the perturbation amplitude. Keep in mind that a larger γ indicates a larger perturbation via flipping the state of more elements in the spiking input.

We first analyze the impact of γ on the attack success rate and perturbation amplitude, as shown in Figure 5.10. A similar conclusion as observed in Section 5.4.2 also holds, that the target attack is more difficult than the untargeted attack. As γ decreases,

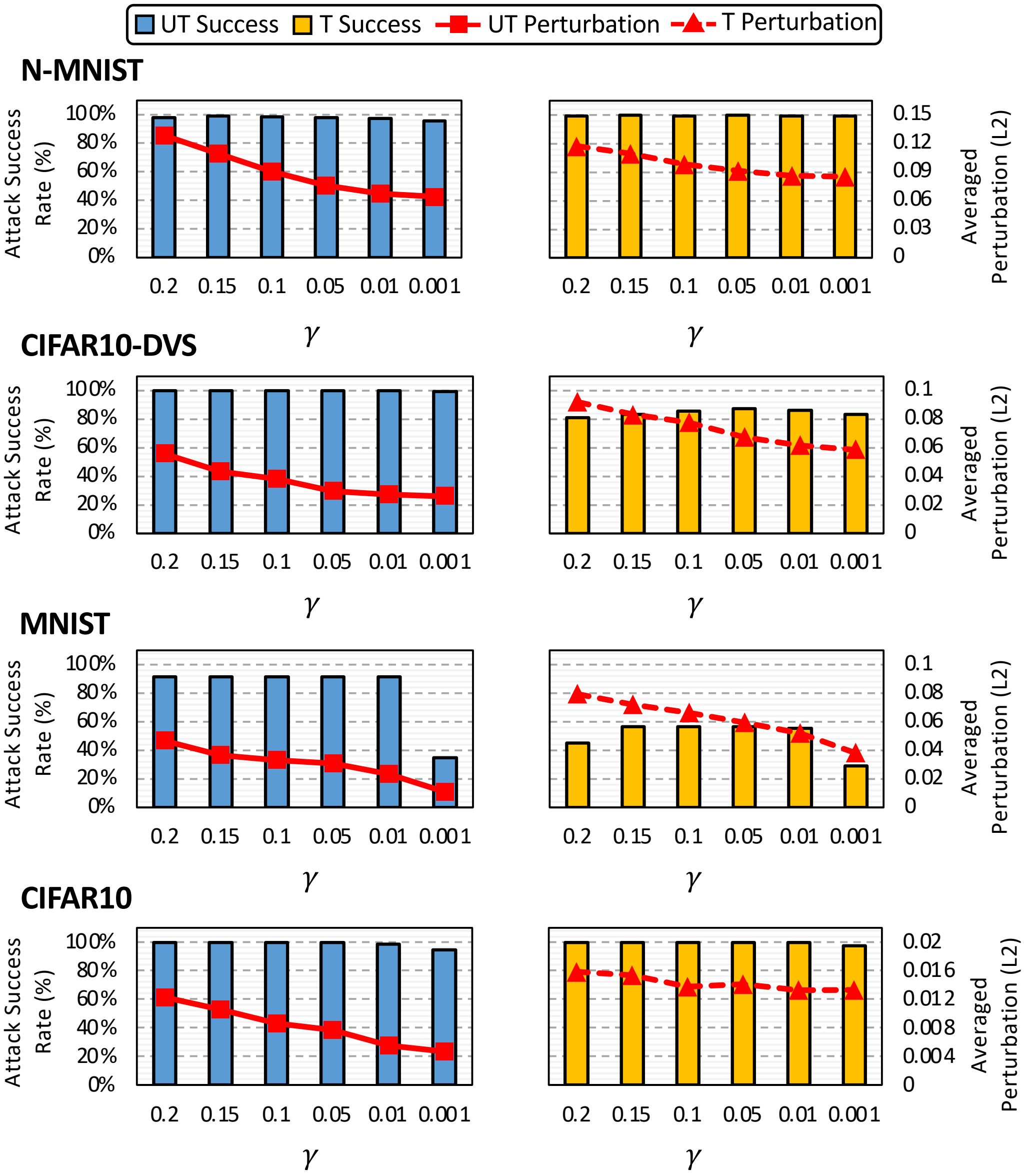


Figure 5.10: Attack success rate and average perturbation with different γ settings. "T" and "UT" refer to targeted attack and untargeted attack, respectively.

the number of elements with flipped state is reduced, leading to smaller perturbation. Whereas, the impact of γ on the attack success rate depends heavily on the attack scenario and the dataset. For the easier untargeted attack, it seems that a slightly large γ is already helpful. The attack success rate will be saturated close to 100% even if at $\gamma = 0.01$. For the targeted attack with higher difficulty, it seems that there exists an obvious peak success rate on these datasets where the γ value equals 0.05. The results are reasonable since the impact of γ is two-fold: i) a too large γ will result in a large perturbation amplitude and might cause a non-convergent attack; ii) a too small γ cannot move the model out of the region with gradient vanishing.

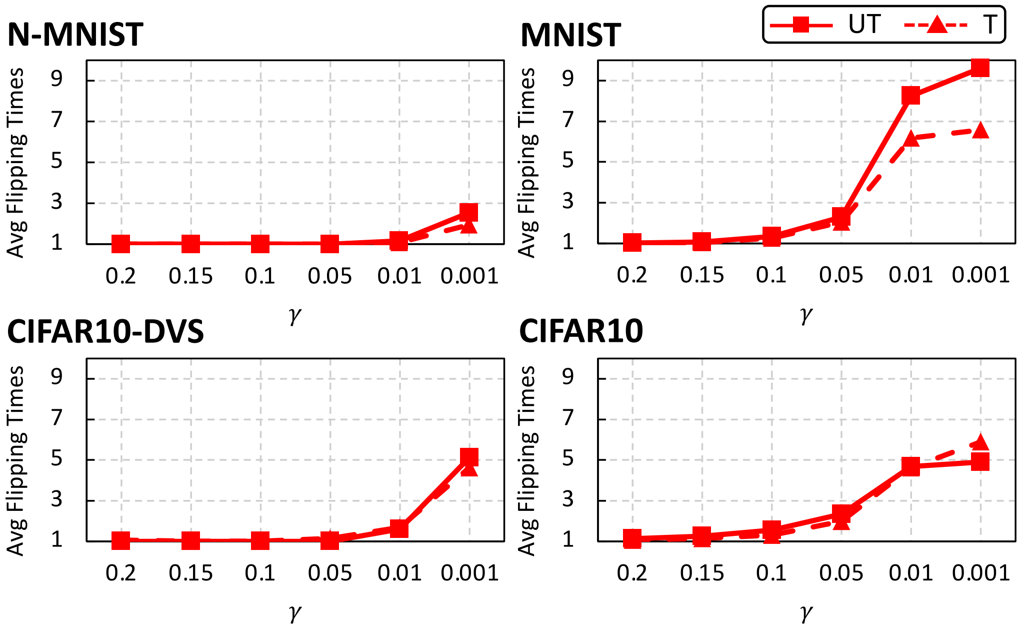


Figure 5.11: Flipping times with different γ settings in RSF. "T" and "UT" refer to targeted attack and untargeted attack, respectively. A smaller γ increases the flipping times since the perturbation is not strong enough to push the model out of the gradient vanishing region.

We also record the number of flipping times under different γ setting, as shown in Figure 5.11. Here the "flipping times" means the number of iterations during the attack process where the gradient vanishing occurs and the spike flipping is needed. When

γ is large, the number of flipping times can be only one since the perturbation is large enough to push the model out of the gradient vanishing region. As γ becomes smaller, the required number of flipping times becomes larger. In order to balance the attack success rate (see Figure 5.10) and the flipping time (see Figure 5.11), we finally recommend the setting of $\gamma = 0.05$ in RSF on the datasets we tested.

5.4.4 Influence of Loss Function and Firing Threshold

Additionally, we evaluate the influence of different training loss functions on the attack success rate. The comparison is summarized in Table 5.4.4. Here the G2S converter and RSF are switched on. The model trained by CE loss leads to a lower attack success rate compared to the one trained by MSE loss, and the gap is especially large in the targeted attack scenario. As explained in Section 5.3.1, this reflects the "trap" region of the models trained by CE loss due to the the increasing spike activities in the penultimate layer during attack.

	MSE Loss		CE Loss	
Dataset	UT	T	UT	T
N-MNIST	97.38%	99.44%	90.12%	16.78%
CIFAR10-DVS	100%	86.35%	100%	82.95%
MNIST	91.31%	55.33%	93.16%	47.81%
CIFAR10	98.68%	99.72%	98.48%	40.51%

Table 5.7: Impact of the loss function on the attack success rate (without firing threshold optimization). "T" and "UT" refer to targeted attack and untargeted attack, respectively.

To improve the attack effectiveness, we increase the firing threshold of the penultimate layer during attack to reduce the spiking activities. Note that we only modify the penultimate layer's firing threshold in the forward pass during the generation of adversarial examples. The experimental results are provided in Figure 5.12. For untargeted attack, the increase of the firing threshold can improve the attack success rate to almost 100% on all datasets. For targeted attack, the cases present different behaviors. Specifically, on image datasets (i.e., MNIST and CIFAR10), the attack success rate can be quickly improved and remained at about 100%; while on spiking datasets (i.e., N-MNIST and CIFAR10-DVS), the attack success rate initially goes higher and then decreases, in other words, there exists a best threshold setting. This might be due to the sparse-event nature of the neuromorphic datasets, on which the number of spikes injected into the last layer will be decreased severely if the firing threshold becomes large enough, leading to a fixed loss value and thus a degraded attack success rate. Moreover, from the perturbation distribution, it can be seen that the increase of the firing threshold does not introduce much extra perturbation in most cases. All the above results indicate that appropriately increasing the firing threshold of the penultimate layer is able to improve the attack effectiveness significantly without enlarging the perturbation.

In Figure 5.12(a), we do not strictly bound ϵ, in order to avoid disturbing the analysis of the firing threshold. The average perturbation magnitude values are shown in Figure 5.12(b), which are relatively small (within 0.08 in most cases). We further analyze the attack success rate under the limitation of strict perturbation bounds. Specifically, during attack iterations, if the average perturbation per point is greater than a pre-defined value ϵ, the attack is considered as a failure. As shown in Figure 5.12(c), our attack method can still achieve a considerable attack success rate with $\epsilon = 0.08$ when compared to the results in Figure 5.12(a) for most cases. While there is a degradation for targeted attack over the N-MNIST dataset, which may be caused by the high sparsity of the spike inputs in that dataset.

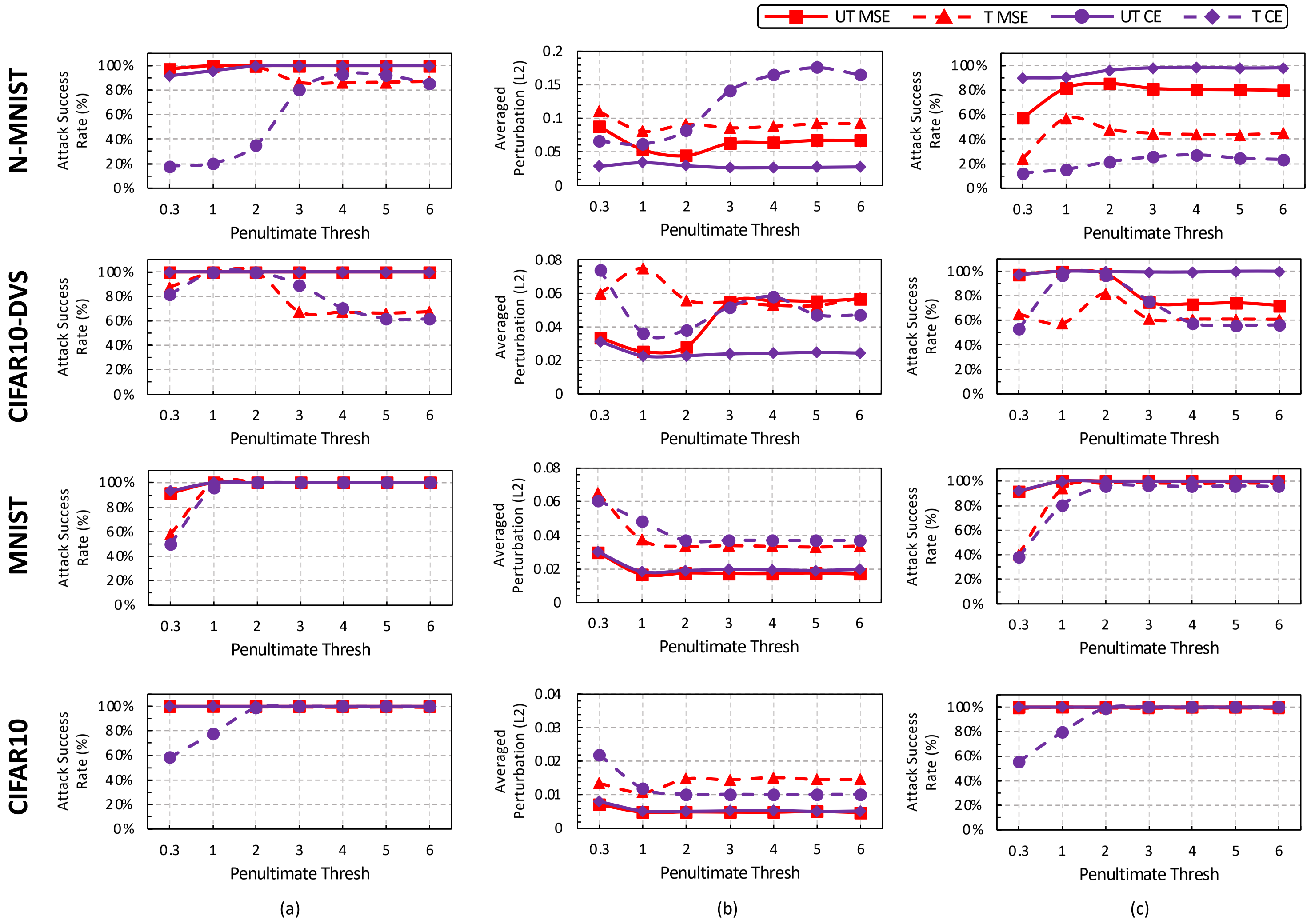


Figure 5.12: Attack effectiveness with different firing threshold: (a) attack success rate without strict ϵ bound; (b) average perturbation without strict ϵ bound; (c) attack success rate under strict perturbation bound ($\epsilon = 0.08$). “T” and “UT” refer to targeted attack and untargeted attack, respectively. In most cases, we can achieve a high attack success rate and acceptable perturbation with a slightly larger firing threshold at the penultimate layer, even if with strict perturbation bound ($\epsilon = 0.08$).

5.4.5 Effectieveness Comparison with Existing SNN Attack

As discussed in Section 5.1.3, our attack is quite different from previous work using trial-and-error input perturbation [107, 71] or SNN/ANN model conversion [70]. Beyond the methodology difference, here we coarsely discuss the attack effectiveness. Due to the high complexity of the trial-and-error manner, the testing dataset is quite small (e.g., USPS dataset [107]) or even with only one single example [71]. In contrast, we demonstrate the effective adversarial attack on much larger datasets. For the SNN/ANN model conversion method [70], the authors show results on the CIFAR10 dataset. In that work, the authors used the accuracy loss of the model, which is caused by substituting the original inputs with the adversarial examples, for the evaluation of the attack effectiveness. We compare our attack results with theirs (inferred from the figure data in [70]) on CIFAR10 under different ϵ configurations, as shown in Table 5.4.5. It can be seen that our attack method can incur more model accuracy loss in most cases, which indicates our better attack effectiveness.

ϵ	8/255	16/255	32/255	64/255
Untargeted [70]	37.50%	62.50%	75.00%	77.00%
Untargeted (ours)	50.47%	72.46%	76.67%	76.86%
Targeted [70]	20.00%	37.50%	52.50%	63.00%
Targeted (ours)	19.16%	42.36%	65.58%	71.48%

Table 5.8: Comparison of the accuracy loss between our work and prior work [70] under different perturbation bounds.

5.4.6 Other Gradient Based Attack Methods and Datasets

In this subsection, we validate the effectiveness of the proposed attack methodology using more experiments with advanced attack methods (e.g., CWL2 [42]) and dynamic datasets (e.g., Gesture-DVS [112]).

CWL2 is an advanced adversarial attack method that is widely applied in ANN attack, which integrates a regularization item to restrict the magnitude of the perturbation. We tailor Algorithm 5 to perform CWL2 attack against SNN models over image-based inputs. The adversarial example generation follows

$$xi'_{k+1} = xi'_k + \delta i_k - c \times \nabla_{xi'_k} \|xi'_k - xi'_0\|_2^2, \tag{5.9}$$

where xi'_0 and xi'_k represent the original input and the adversarial example generated at the kth attack iteration. c is a parameter that determines the impact of regularization item. A larger c indicates smaller perturbation at the cost of possibly lower attack success rate. The CWL2 attack would degrade to the classic BIM attack when $c = 0$.

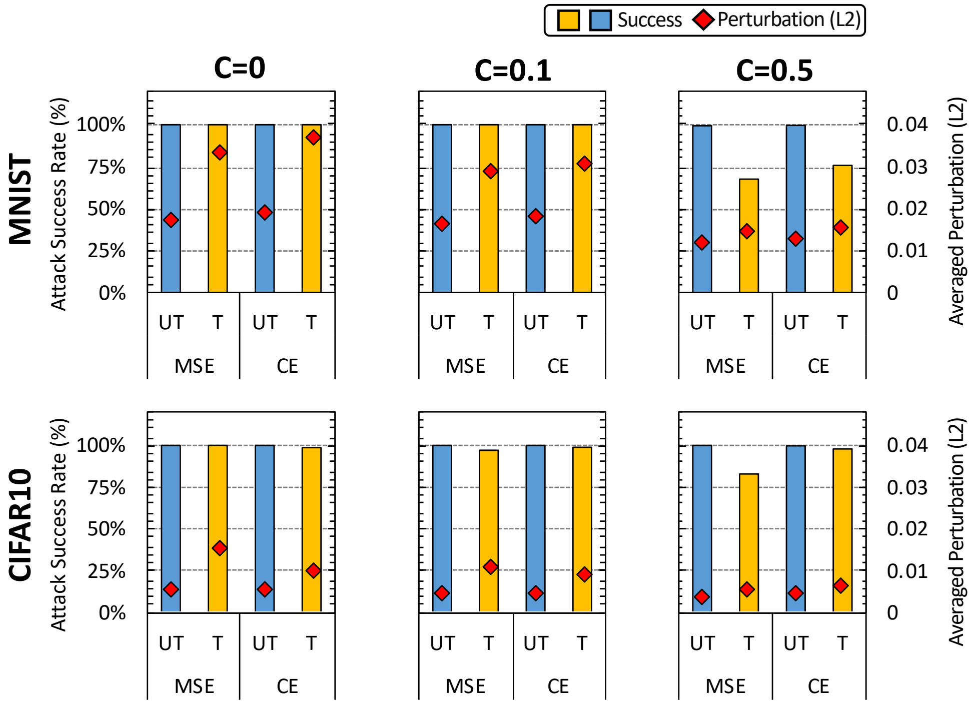


Figure 5.13: Attack effectiveness with CWL2 under different settings of c. A larger c indicates a smaller perturbation but may compromise the attack success rate.

We tested the tailored SNN-oriented CWL2 attack on MNIST and CIFAR10 datasets with different configurations of c. As illustrated in Figure 5.13, a slight increase of c

$(c = 0.1)$ helps reduce the perturbation magnitude without sacrificing the attack success rate, compared to the results at $(c = 0)$. However, when c is too large $(c = 0.5)$, the attack success rate decreases. For example, the targeted attack success rate on MNIST dataset is reduced by up to 32.45% when $c = 0.5$.

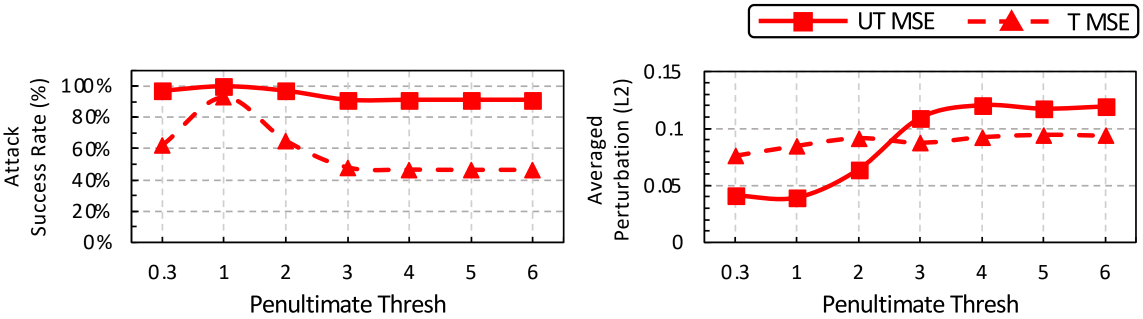


Figure 5.14: Attack effectiveness on the Gesture-DVS dataset.

In addition, we also applied our attack method on Gesture-DVS. The model configurations have been shown in Table 5.4.1. We only test the cases with MSE training loss function for simplicity, and the attack results are shown in Figure 5.14. Our methodology can still achieve a high attack success rate with acceptable perturbation even on this dynamic dataset. The trend of attack success rate variation under different penultimate layer threshold setting is similar to that on other spike-based datasets we have tested earlier.

5.4.7 SNNs VS. ANNs Against Adversarial Attack

In this subsection, we further compare SNNs and ANNs against adversarial attack. In essence, we make the comparison from two perspectives: the perturbation distance demanded for successful attack; the transferability between the adversarial examples of ANNs and SNNs. Here, the transfereability of model A's adversarial examples on model B represents the attack success rate of attacking model B with the adversarial examples generated by model A. In our evaluation, a larger perturbation distance indicates more challenges to attack a model; a lower transferability implies that the adversarial examples generated by one model are lower possible to attack other models successfully.

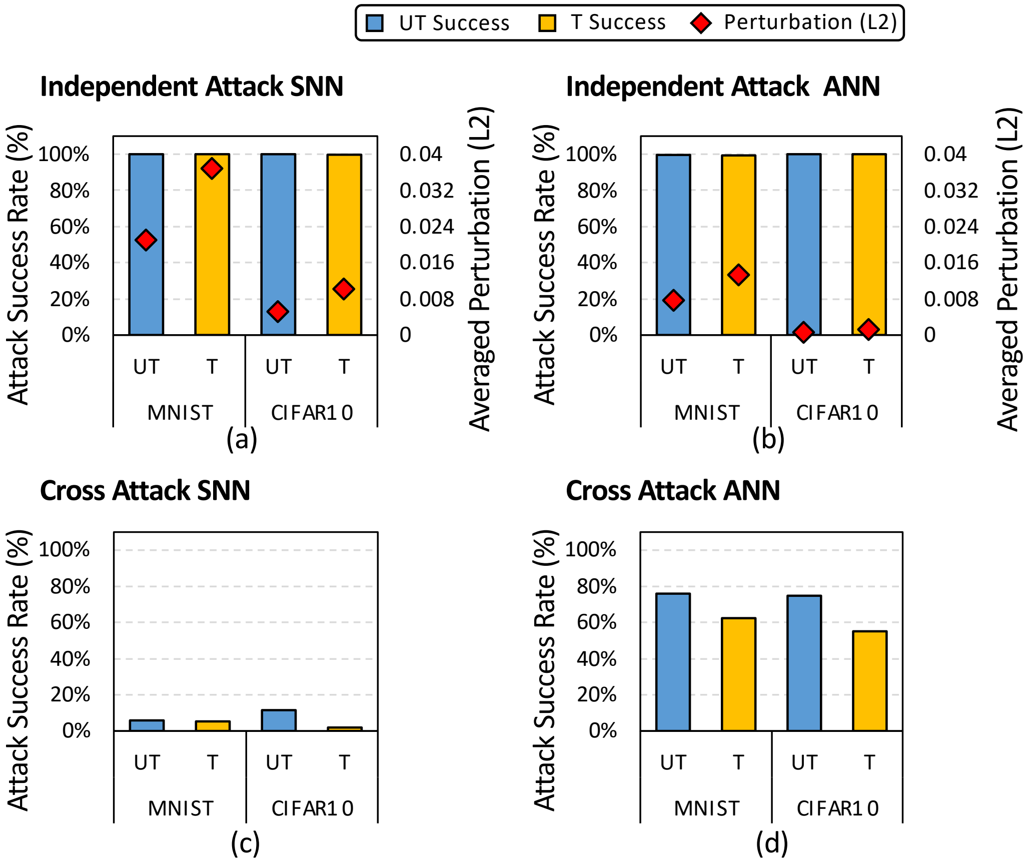


Figure 5.15: Attack success rate comparison between ANNs and SNNs under gradient-based attack. "T" and "UT" refer to targeted attack and untargeted attack, respectively. (a)-(b) Independent attack; (c)-(d) Cross attack. In this scenario, attacking SNNs requires larger perturbation than attacking ANNs and the adversarial examples generated by attacking the ANN models fail to attack the SNN models.

In this subsection, we select image-based datasets, MNIST and CIFAR10s. For ANN models, we use the same network structure as SNN models given in Table 5.4.1. The training loss function is CE here. We test two attack scenarios: independent attack and cross attack. For the independent attack, the ANN models are attacked using the BIM method in Equation (5.3); while the SNN models are attacked using the proposed gradient-based method and also a gradient-free method. Note that the firing threshold of the the penultimate layer of SNN models during attack is set to 2 in this subsection as suggested by Figure 5.12. For the cross attack, we use the adversarial examples generated by attacking the SNN models to mislead the ANN models, or vice versa.

From Figure 5.15(a)-(b), we can easily observe that all attack success rates are quite high in the independent attack scenario. While, attacking the SNN models requires larger perturbation than attacking the ANN models in the above experiment. From the results of the cross attack in Figure 5.15(c)-(d), we find that using the adversarial examples generated by attacking ANN models to fool the SNN models is very difficult, with only <12% success rate, indicating the lower transferability of the ANN adversarial examples. The observation in this scenario reflects that SNN adversarial examples are easier to transfer to an ANN model with the same network structure.

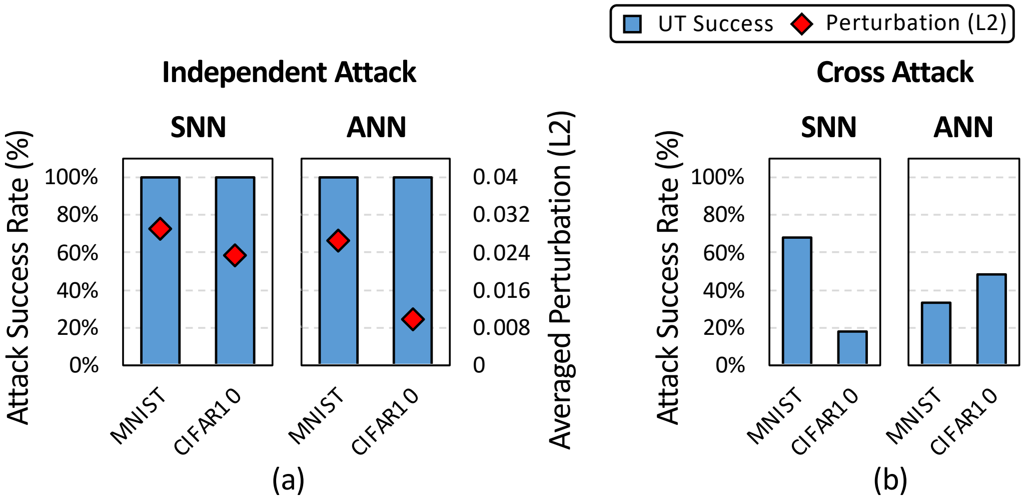


Figure 5.16: Attack success rate comparison between ANNs and SNNs under untargeted boundary attack. (a) Independent attack; (b) Cross attack. In this scenario, attacking SNNs still requires larger perturbation than attacking ANNs; however, the ANN adversarial examples on MNIST present better transferability than the SNN adversarial examples.

Besides the gradient-based attack method, we also exam the gradient-free boundary attack method [43]. Since the computational complexity of boundary attack is much higher than the gradient-based attack, we only evaluate the untargeted attack. The results are depicted in Figure 5.16. We find that the required perturbation to attack SNNs is still higher. However, under cross attack, the ANN adversarial examples present better transferability than the SNN adversarial examples, which indicates the SNN model in this scenario is easier to attack.

5.5 Conclusion

SNNs have attracted broad attention and have been widely deployed in neuromorphic devices due to the importance for brain-inspired computing. Naturally, the security problem of SNNs should be considered. In this Chapter, we first identify the challenges in attacking an SNN model with spatiotemporal-gradient-based methods, including the incompatibility between the spiking inputs and the continuous gradients, and the gradient vanishing problem. Second, we design a gradient-to-spike (G2S) converter and a restricted spike flipper (RSF) to address the mentioned two challenges, respectively. Our methodology can control the perturbation amplitude well and is applicable to both spiking and image data formats. Interestingly, we find that there is a "trap" region in SNN models trained by CE loss, which can be overcome by adjusting the firing threshold of the penultimate layer. We conduct extensive experiments on various datasets and show 99%+ attack success rate in most cases, which is the best result on SNN attack. Furthermore, we compare the attack of SNNs and ANNs. From our empirical results, the adversarial examples for SNNs require a larger perturbation distance, but it still remains open whether SNNs can be more robust than ANNs against adversarial attack.

For future work, we recommend several interesting topics. Although we only study the white-box adversarial attack to avoid shifting the focus of presenting our methodology, the black-box adversarial attack should be investigated because it is more practical. Fortunately, the proposed methods in this work can be transferred into the black-box attack scenario. Second, we only analyze the influence of loss function and firing threshold due to the page limit. It still remains an open question that whether other factors can affect the attack effectiveness, such as the gradient approximation form of the firing activities, the time window length for rate coding or the coding scheme itself, the network structure, and other solutions that can substitute G2S and RSF. Third, more appropriate evaluation metrics should be designed to evaluate the perturbation for spike data. Fourth, a more comprehensive research to compare the robustness between ANNs and SNNs against adversarial attack is an interesting topic. Fifth, the attack against physical neuromorphic devices rather than just theoretical models is more attractive. At last, compared to the attack methods, the defense techniques are highly expected for the construction of large-scale neuromorphic systems.

Chapter 6

Toward Robust Spiking Neural Networks Against Adversarial Perturbation

As spiking neural networks (SNNs) are deployed increasingly in real-world efficiency critical applications, the security concerns in SNNs attract more attention. In previous Chapter we demonstrate an SNN can be attacked with adversarial examples. How to build a robust SNN becomes an urgent issue. Recently, many studies apply certified training in artificial neural networks (ANNs), which can improve the robustness of an NN model promisely. However, existing certifications cannot transfer to SNNs directly because of the distinct neuron behavior and input formats for SNNs. In this Chapter, we first design S-IBP and S-CROWN that tackle the non-linear functions in SNNs' neuron modeling. Then, we formalize the boundaries for both digital and spike inputs. Finally, we demonstrate the efficiency of our proposed robust training method in different datasets and model architectures.

6.1 Overview and Preliminaries

With more attention to the study of SNNs, the security issues raise concerns in the community. Adversarial attack [37, 42, 113, 114] is one of the most intuitive ways to evaluate the robustness of a model. In adversarial attacks, the attacker generates adversarial examples to fool a model predicting wrong. Currently, SNNs are demonstrated can be attacked through adversarial examples [70, 69, 72]. It is urgent to explore an efficient way to improve the robustness of SNN models.

Previously, researchers have investigated the impact of hyper-parameter selection [73] and input filtering [72] on the adversarial attack in SNNs. However, these methods do not directly promote the classification behavior of a given SNN model. Unlike SNNs, how to improve the robustness of an artificial neural network (ANN) is well studied. Recently, trianing a neural network model with certified defense methods [115, 116, 46, 47] show remarkable guarantee to improve the model's robustness. CROWN-IBP [46] is one of the most promising certified training methods with polynomial computianal cost compared with natural training. The CROWN-IBP method will compute the output boundary for a given bounded input. The core mission in CROWN-IBP certified training is to find the upper and lower bound function for each operation and find tight linear relaxation for non-linear operations. However, the current CROWN-IBP method cannot be directly applied to SNNs. Firstly, the neuron dynamic in SNNs is more complicated. Hence, some new boundary functions should be defined to bound the unique non-linear operations in SNNs. Secondly, SNNs accept both spike and digital inputs, which requires additional boundary generalization for different input types.

Enlightened by certification training in ANNs, in this work, we designed an end-to-end robust training method to improve the robustness of an SNN model against adversarial attacks. Specifically, our major contributions can be summarized as follows:

- We design S-IBP and S-CROWN to tackle the non-linear fire function and temporal update in SNNs which firstly introduce SNNs to the linear relaxation based verification family.
- We formalized ℓ_0-norm and ℓ_∞-norm boundaries for spike and digital inputs, respectively.
- Our proposed methods are evaluated on MNIST [79], FMNIST [117] and NMNIST [82] datasets. The experinetal results show that we can achieve a maximum 37.7% attack error reduction with 3.7% original accuracy loss.

6.1.1 Input Format

In this Chapter, we focus on the image recognition tasks. The input of an SNN can be spike events captured by dynamic vision sensors [118], which naturally fits the input layer of an SNN. Also, SNNs can take a digital image as input, however, sampling should be involved before feeding to the SNN. Specifically, we adopt Bernoulli sampling [3] for digital inputs that follows

$$P(\dot{x}_t[i] = 1) = \hat{x}[i]. \tag{6.1}$$

We use $\dot{x}_t$ to represent the spike input at time step t. $\hat{x}$ is the digital input after normalization to $[0, 1]$ for each pixel. For a digital input, the probability that a pixel i in the corresponding spike input has a spike event equals to its normalized gray value. The example of spike and digital inputs are shown in Figure 5.2.

6.1.2 Adversarial Attack in SNN

In this Chapter we adopt the utargeted white-box gradient-based attack that proposed in Chapter 5. The framework of the adversarial attack in SNNs is shown in Figure 6.1.

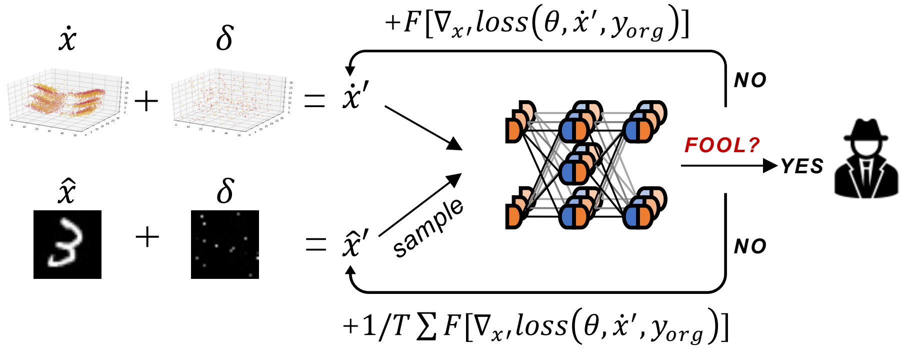

$\dot{x}$
δ
$+$
$= \dot{x}'$
$+F[\nabla_{x'} loss(\theta, \dot{x}', y_{org})]$
NO
FOOL?
YES
NO
$\hat{x}$
δ
$+$
$= \hat{x}'$
sample
$+1/T \sum F[\nabla_{x'} loss(\theta, \dot{x}', y_{org})]$

Figure 6.1: The framework of untargeted white-box gradient-based attack on SNN.

In adversarial attack, the attacker adds imperceptible noise on the input to fool the model generating an incorrect prediction result. In untargeted attack senario, the model predicts the adversarial example as any other class except the ground-truth, which can be formulated as

$$\underset{\delta}{\operatorname{argmin}} \|\delta\|_p, \ \ s.t. \ f(x+\delta) \neq f(x). \tag{6.2}$$

Here, $f(\cdot)$ is the prediction result of an SNN model, x is the original input, and δ is the noise added on the input. In most cases, it is easier to find an adversarial example if the attacker knows the models, i.e., the parameters and the structure of the model. The adversarial attacks that know all of the model information are called white-box attacks.

The gradient-based attacks are the most efficient ways to generate adversarial examples. In SNNs, the gradient-based attack for the spike and digital inputs can be formulated as

$$\begin{cases} \dot{x}'_{n+1} = \dot{x}'_n + F[\nabla_{\dot{x}'_n} L(\theta, \dot{x}'_n, y_{org})], & \text{spike input,} \\ \\ \hat{x}'_{n+1} = \hat{x}'_n + \dfrac{\sum_t F[\nabla_{\dot{x}'_n} L(\theta, \dot{x}'_n, y_{org})]}{T}, & \text{digital input.} \end{cases} \tag{6.3}$$

We use $\dot{x}'$ and $\hat{x}'$ to represent the spike and digital adversarial examples, i.e., $x' = x + \delta$. L and θ are the loss function and parameters of the model, respectively. The adversarial example is constructed by adding the gradient of inputs with the original label (y_{org}) in loss function. F is a filter function that samples, clips, and generates spike compatible noise. For the digital inputs, the binary noises are averaged along the time steps (t) to construct the floating-point noise. During the attack, attackers can compute the adversarial examples iteratively once the current adversarial example cannot fool the model successively. We use n to denote the attack iteration.

6.1.3 Certified Training

Recently, the certified training [45, 46, 47] has been demonstrated to improve the guaranteed robustness of a neural network. In this work, we leverage the CROWN-IBP method [46], one of the state-of-the-art certified training that can achieve tighter bounds in accepctable training cost. Considering a digital input data bounded with ℓ_∞-norm, the goal of CROWN-IBP is to identify whether arbitrary input data within the boundary can fool the model. The CROWN-IBP contains two parts of bounding methods: IBP [45] and CROWN [119].

IBP

In IBP processing, the lower and upper bounds of each layer's feature map are computed along the forward propagation, i.e. start from the input layer to the output layer. During the bound computation, the linear operation can be easily bounded once we know the maximum and minimum values of input. However, the upper and lower bound after the non-linear operations need to be dedicated to analysis. Once we acquire the bounds of output we can evaluate the robustness of the model for the given input purbation set.

CROWN

Unlike IBP, CROWN bounds the model in a backward propagation manner recursively. The goal of CROWN is to formulate the output bounds as a linear equation of input. In order to achieve this goal, every operation in an NN model should be bounded by two linear equations.

Although CROWN can achieve very tight bounds, the computinal cost is remarkably higher than IBP. So CROWN-IBP, by combining the fast IBP bounds in a forward bounding pass and CROWN in a backward bounding pass can efficiently and consistently outperforms IBP baselines on training verifiably robust neural networks.

6.2 Robust Training on SNN

From previous studies, certification training is an efficient methodology to improve the robustness of a neural network model. However, existing methods cannot be directly applied to the SNNs. The main reason is that the non-linear function in SNNs (fire and temporal update) and the input boundary formalization are special in SNNs. In order to achieve certification training in SNNs, we designed S-IBP and S-CROWN to tackle the non-linear neuron behavior in SNNs. Also, we analyzed the input boundary for both spike and digital data.

6.2.1 S-IBP & S-CROWN

As described in Section 6.1.3, the core mission of certification training is to find the bound of IBP and CROWN for each function. The information propagation in SNNs includes $fire$, temporal update and spatial update as shown in Figure 2.3(a). In this subsection, we detail the upper bound and lower bound of S-IBP and S-CROWN for each function.

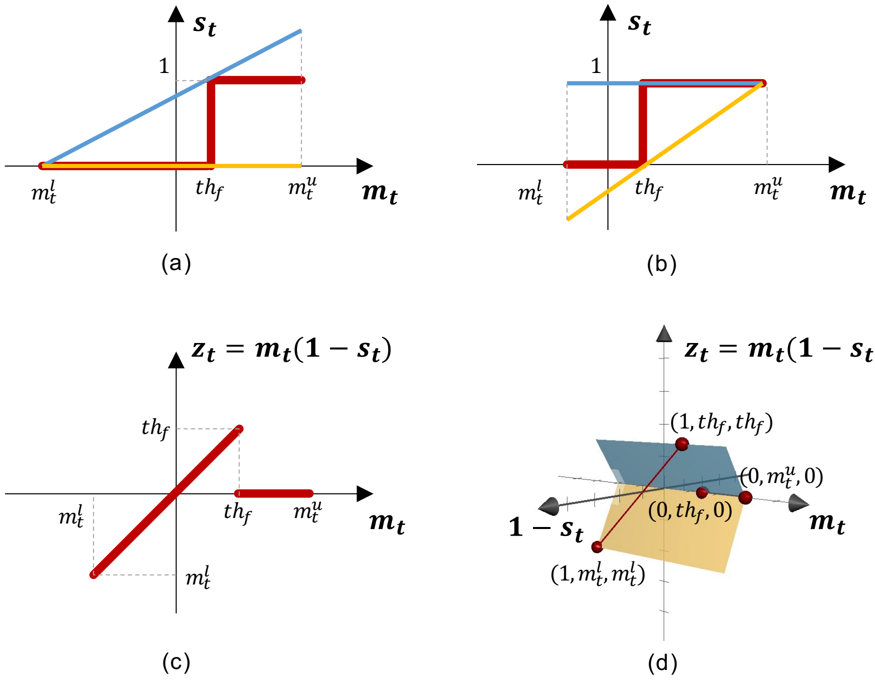
Fire

The $fire$ function describes the relation between the membrane potential m_t and spike s_t of a neuron as Equation 2.1, i.e. once the neuron's membrane potential is greater than a threshold th_f, the neuron will fire a spike. Assume we have already acquired the S-IBP upper bound m_t^u and lower bound m_t^l for membrane potential (note that for symbol m_t^l, l does not indicate the *layer* number, instead l represents the lower bound of an intermediate data), the S-IBP bounds for spike can be calculated with

$$s_t^u = fire(m_t^u - th_f), \quad s_t^l = fire(m_t^l - th_f). \tag{6.4}$$

During S-CROWN, we need to find two linear equations to bound the $fire$ function. When the S-IBP upper bound of membrane potential m_t^u is smaller than the threshold th_f, we can conclude that the neuron must not fire. Also, when the S-IBP lower bound m_t^l is greater than the threshold, we can make sure the neuron must fire. Thus, we mainly need to consider the unstable case, i.e., $m_t^l < th_f \leq m_t^u$. In order to achieve the lowest bound relaxation, we design two boundary systems as Figure 6.2(a) and (b). We use the red line to represent the $fire$ function. The blue and yellow lines represent the boundary functions. In our design, when the S-IBP lower bound of membrane potential m_t^l far smaller than th, we set the S-CROWN lower bound to $s_t = 0$ and the S-CROWN upper bound is a line that crosses $(m_t^l, 0)$ and $(th, 1)$. On the contrary, when m_t^u is far larger than th, we set the S-CROWN upper bound to $s_t = 1$ and the S-CROWN lower bound is a line that passes $(th_f, 0)$ and $(m_t^u, 1)$. Overall, the S-CROWN boundary for the $fire$ function under different cases can be summarized as

$$
\begin{cases}
0 \le s_t \le 0, & m_t^u < th_f, \\
1 \le s_t \le 1, & m_t^l \ge th_f, \\
0 \le s_t \le \frac{m_t - m_t^l}{th_f - m_t^l}, & 0 \le m_t^u - th_f < th_f - m_t^l \\
\frac{m_t - th_f}{m_t^u - th_f} \le s_t \le 1, & m_t^u - th_f \ge th_f - m_t^l > 0
\end{cases}
\tag{6.5}
$$



(a)

(b)

(c)

(d)

Figure 6.2: S-CROWN upper and lower bounds for the *fire* function when (a) m_t^l far smaller than th and (b) m_t^u far larger than th. (c) Non-linear temporal part of memory potential update. (d) S-CROWN upper and lower bounds for the temporal part of memory potential update (unstable case).

Temporal Update

According to Equation 2.2, the temporal update of membrane potential can be formalized as $\alpha m_t(1 - s_t)$. Since α is a constant factor, in this subsection we do not include it in the boundary analysis. Assume $z_t = m_t(1 - s_t)$, and the S-IBP upper and lower bound for membrane potential at time step t has been acquired. The relation between m_t and z_t can be described through Figure 6.2(c). Based on the LIF model, the membrane potential will be reset to 0 once its value acrosses the pre-defined threshold th. Thus, the S-IBP bounds for z_t can be computed with

$$\begin{cases} z_t^u = m_t^u, \;\; z_t^l = m_t^l, & m_t^u < th_f, \\ z_t^u = z_t^l = 0, & m_t^l \geq th_f, \\ z_t^u = th_f, \; z_t^l = min(0, m_t^l), & m_t^l < th_f \leq m_t^u. \end{cases} \tag{6.6}$$

Ituitivly, when $m_t^u < th_f$ or $m_t^l \geq th_f$, there is no boundary relaxation in S-IBP. We only need to care about the boundary when $m_t^l < th_f \leq m_t^u$ (unstable case). From Fgiure 6.2(c) we can find that the $z_t \in [min(0, m_t^l), \;\; th_f)$ for unstable case.

During S-CROWN, z_t can be also bounded without relaxation when $m_t^u < th_f$ and $m_t^l \geq th_f$. The S-CROWN upper bound (blue plane) and lower bound (yellow plane) for z_t when $m_t^l < th_f \leq m_t^u$ is shown in Figure 6.2(d). Here we use an additional $(1 - s_t)$ axis to help us build the boundary. Note that the membrane potential is related to the spike status. Once $s_t = 1 \rightarrow (1 - s_t) = 0$, m_t must greater than th_f and $z_t = 0$. When $s_t = 0 \rightarrow (1 - s_t) = 1$, m_t must smaller than th_f and $z_t = m_t$. Thus we need to find two planes to bound these two function (red lines in Figure 6.2(d)). In summary, the S-CROWN boundary for the temporal update can be formulated as

$$\begin{cases} m_t \leq z_t \leq m_t, & m_t^u < th_f, \\ 0 \quad \leq z_t \leq 0, & m_t^l \geq th_f, \\ (1 - s_t) m_t^l < z_t < (1 - s_t) th_f, & m_t^l < th_f \leq m_t^u. \end{cases} \tag{6.7}$$

Thus, in our design, the S-CROWN boundaries for the temporal update are the functions of s_t under the unstable case.

Spatial Update

The spatial update in SNNs is shown in Equation 2.2 and 2.3. Similar to ANNs, the spatial update in SNNs is composed of CONV/FC/POOL (in this work we focus on the CONV and FC). In SNNs CONV/FC takes spike events and weight as input. Since the spike events are in binary format, the S-IBP of spatial update can be implemented with

$$\begin{cases} center = w[k] \otimes s_t^l[k] + b[k], \\ sp_t^u[k+1] = center + w[k]^+ \otimes (s_t^l[k] = 0 \cap s_t^u[k] = 1), \\ sp_t^l[k+1] \;= center + w[k]^- \otimes (s_t^l[k] = 0 \cap s_t^u[k] = 1). \end{cases} \tag{6.8}$$

We use sp_t to represent the result of the spatial update. $\otimes$ denotes CONV/FC operation. In Equation 6.8, $s_t^l[k] = 1$ represents those pre-synaptic neurons in layer k who are must fire. The stable fired pre-synaptic neurons contribute the same for both S-IBP upper and lower bound of sp_t. The unstabled pre-synaptic neurons can be represented with $(s_t^l[k] = 0 \cap s_t^u[k] = 1)$, whose upper and lower bound for spike status are 1 and 0. These unstable pre-synaptic neurons will perform CONV/FC with the positive and negative weights to affect the S-IBP upper and lower bound of sp_t.

Since the spatial update is a linear operation, the S-CRWON for spatial update does not have relaxation, which is the same as the case in ANNs. During the S-CROWN phase, we do not need to design boundary functions to bound the spatial update in SNNs.

6.2.2 Input Boundary Formalization

In SNNs, except for the distinct non-linear behavior of information propagation, the input layer of an SNN model only accept binary spike. The special data format for the input layer leads to different boundary formalizations for spike and digital images.

Spike Input

An example of spike input is shown in Figure 5.2. All elements in a spike data are in binary format which is compatible with the input layer of an SNN model. For each element $\dot{x}_t[i]$ in a spike input, the boundary of that element can be either stable cases: $\dot{x}_t^u[i] = \dot{x}_t^l[i] = 0$; $\dot{x}_t^u[i] = \dot{x}_t^l[i] = 1$ or unstable case: $\dot{x}_t^u[i] = 1, \dot{x}_t^l[i] = 0$. For a spike input with uncertainty noise, we can only control how many data points in the spike input are unstable. Thus, the boundary for a spike input can be formulated with ℓ_0-norm. Specifically, we can pick $size(\dot{x}) \times \epsilon$ elements from a spike input and set them as unstable points. Also, the ℓ_0-norm boundary can be interpreted as the probability of an element is unstable, which can be formulated as

$$P(\dot{x}_t^u[i] = 1, \dot{x}_t^l[i] = 0) = \epsilon \;\Leftrightarrow\; |\dot{x}' - \dot{x}|_0 \leq size(\dot{x}) \times \epsilon \tag{6.9}$$

Here, $\dot{x}'$ is an arbitrary adversarial example that has at most $size(\dot{x}) \times \epsilon$ data points different from $\dot{x}$.

Note that we can only certify the robustness of a spike input after we have picked the unstable data points. Under our robustness formulation, we cannot guarantee the robustness of a spike input under a given ℓ_0-norm. The reason is that the search space for ℓ_0-norm cannot be bounded.

Digital Input

Unlike spike inputs, digital inputs need an additional Bernoulli sampling before feeding the data to the input layer as Equation 6.1. After the sampling, the digital input $\hat{x}$ is converted to a spike input $\dot{x}$ which fits the input layer of an SNN. Note that we have

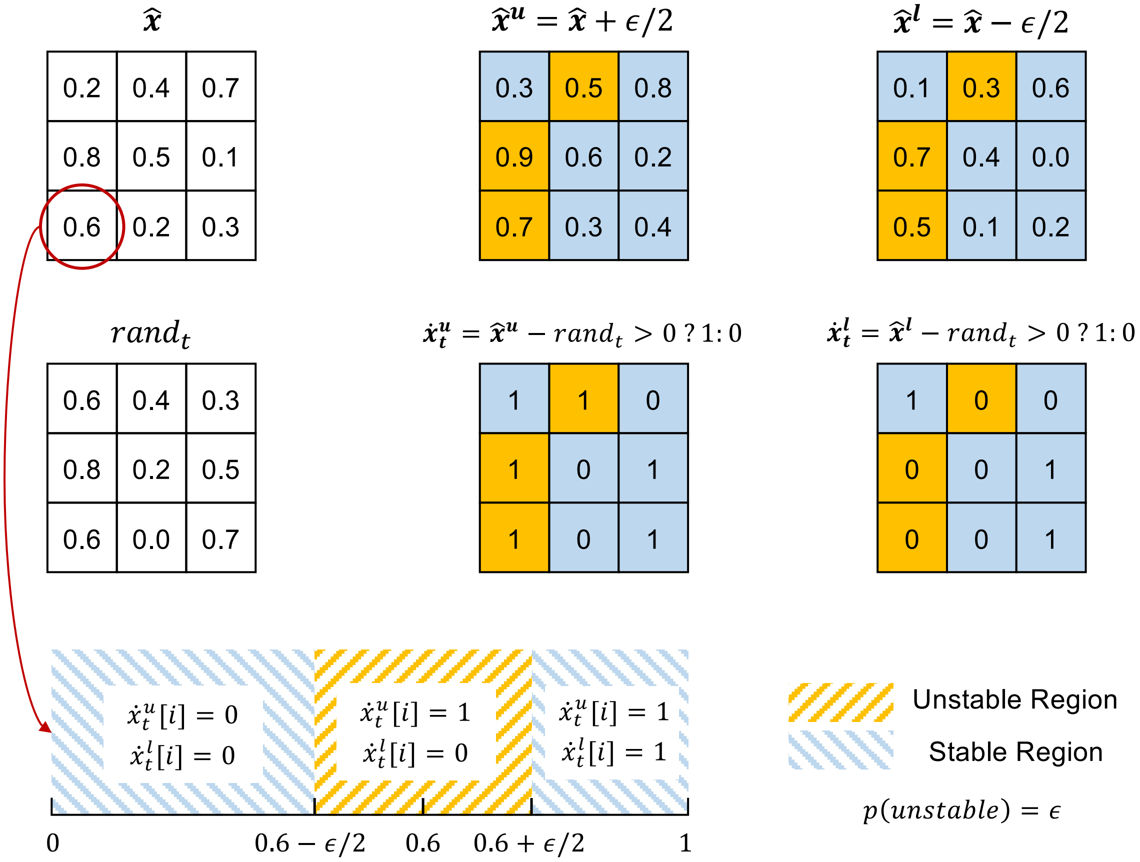

$\widehat{\boldsymbol{x}}$
0.2 0.4 0.7
0.8 0.5 0.1
0.6 0.2 0.3
$\widehat{\boldsymbol{x}}^{\boldsymbol{u}} = \widehat{\boldsymbol{x}} + \epsilon/2$
0.3 0.5 0.8
0.9 0.6 0.2
0.7 0.3 0.4
$\widehat{\boldsymbol{x}}^{\boldsymbol{l}} = \widehat{\boldsymbol{x}} - \epsilon/2$
0.1 0.3 0.6
0.7 0.4 0.0
0.5 0.1 0.2
$rand_t$
0.6 0.4 0.3
0.8 0.2 0.5
0.6 0.0 0.7
$\dot{\boldsymbol{x}}_t^{\boldsymbol{u}} = \widehat{\boldsymbol{x}}^{\boldsymbol{u}} - rand_t > 0\ ?\ 1 : 0$
1 1 0
1 0 1
1 0 1
$\dot{\boldsymbol{x}}_t^{\boldsymbol{l}} = \widehat{\boldsymbol{x}}^{\boldsymbol{l}} - rand_t > 0\ ?\ 1 : 0$
1 0 0
0 0 1
0 0 1
$\dot{x}_t^u[i] = 0$
$\dot{x}_t^l[i] = 0$
$\dot{x}_t^u[i] = 1$
$\dot{x}_t^l[i] = 0$
$\dot{x}_t^u[i] = 1$
$\dot{x}_t^l[i] = 1$
0
$0.6 - \epsilon/2$
0.6
$0.6 + \epsilon/2$
1
Unstable Region
Stable Region
$p(unstable) = \epsilon$

Figure 6.3: Relation between ℓ_∞-norm digital input doundary and ℓ_0-norm spike input boundary. In this example, $\epsilon = 0.2$.

defined the ℓ_0-norm boundary for a spike input. We need to further explore how to define a boundary for a digital input when its sampled data is bounded with ℓ_0-norm. Based on our analysis, we find that the corresponding digital input boundary can be formalized with ℓ_∞-norm as Figure 6.3. We use $rand_t$ to represent a rand mask to achieve Bernoulli sampling for each time step:

$$\dot{x}_t[i] = \begin{cases} 1, & \hat{x}[i] > rand_t[i], \\ 0, & \text{otherwise.} \end{cases} \tag{6.10}$$

After we bound the digital input with ℓ_∞-norm, the upper bound of digital image becomes $\hat{x}^u = \hat{x} + \epsilon/2$ and the lower bound is $\hat{x}^l = \hat{x} - \epsilon/2$. For all elements in $\hat{x}$, the difference between the upper bound and the lower bound is ϵ, i.e., $\hat{x}^u[i] - \hat{x}^l[i] = \epsilon$. For each time step, $\hat{x}^u$ and $\hat{x}^l$ use the identical random map $rand_t$ to sample the corresponding spike inputs $\dot{x}^u_t$ and $\dot{x}^l_t$. Based on the sampling processing, the probability of an element in input layer is unstable can be formulated as

$$P(\dot{x}^u_t[i] = 1, \dot{x}^l_t[i] = 0) = \hat{x}^u[i] - \hat{x}^l[i] = \epsilon. \tag{6.11}$$

The unstable probability here is exactly the same as the case for spike input as Equation 6.9. Thus, from the static perspective, the ℓ_∞-norm boundary for digital input is equivalent to the ℓ_0-norm boundary for spike input. Since ϵ usually very small, we ignore the corner cases, i.e., when $\hat{x}[i]$ is close to 0 or 1 after normalization.

For digital inputs, although we can bound the input with ℓ_∞-norm, we cannot guarantee the robustness of the input under such boundary. The reason is that during inference, the corresponding spike input is generated through sampling. Thus, for an arbitrary input, the possible sampled results are equivalent to the entire space (each element in the spike input can be 1 or 0), which cannot be bounded.

6.2.3 Robust Training Algorithm

In previous subsections, we have analyzed the S-IBP and S-CROWN for each function in SNNs. Also, we formalized the input boundaries for different input formats. Here, we present the end-to-end robust training for an SNN.

Flexible Time Steps: Usually, the time steps of an SNN (trained with BPTT) is 10 to 20, however, it is still large when considering robustness training. We note that for each SNN layer, the parameters (weight and bias) are shared among different time steps. Thus, we can set arbitrary time steps T' for the robust training.

S-IBP: During the robust training, the inputs first pass the S-IBP as Algorithm 6. At the beginning, the input is bounded according to the input type. Then, the intermediate data are bounded along the forward direction. Finally, the upper and lower bound of all intermediate data are stored which will be used during the S-CROWN phase.

S-CROWN: Note that the goal of S-CROWN is to formulate the output bounds of a model as linear equations of input's upper and lower bounds. Also, the boundary is computed from the backward direction. The detailed steps of S-CROWN are shown in Algorithm 7.

In the output layer, the S-CROWN boundary can be formulated as

$$f(\dot{x}) \geq \sum_t \underbrace{I/T'}_{As_t[K]} * s_t[K]. \tag{6.12}$$

Here, I is the identity matrix. We use $As_t[K]$ to represent the linear matrix respect to the output, where t denotes the time step and $s_t[K]$ represents the spike events in the output layer. Suppose we formulate each non-linear operation as $q_t = g(p_t)$, where p_t

```
Algorithm 6: S-IBP
Input:
spike input $\dot{x}$ or digital input $\hat{x}$; input boundary $\epsilon$; robust training time steps $T'$;
Func:
for $t = 1$ to $T'$ do
    // input boundary formalization
    if $\hat{x}$ then
        generate random map $rand_t$;
        $\dot{x}^u_t = (\hat{x} + \epsilon/2) - rand_t \;>\; 0 \;?\; 1 : 0;$
        $\dot{x}^l_t \; = (\hat{x} - \epsilon/2) - rand_t \;>\; 0 \;?\; 1 : 0;$
    else
        Randomly pick $size(\dot{x}_t) \times \epsilon$ elements from $\dot{x}_t$ and label the picked
        elements with $pick_t$;
        $\dot{x}^u_t = \dot{x}^l_t = \dot{x}_t; \quad \dot{x}^u_t[pick_t] = 1; \quad \dot{x}^l_t[pick_t] = 0;$
    end if
    $s^u_t[0] = \dot{x}^u_t; \quad s^l_t[0] = \dot{x}^l_t;$

    inital $m^u_t[k] = m^l_t[k] = 0$ for all layers.
    //S-IBP
    for $k = 1$ to $K$ do
        //spatial update (Equation 6.8)
        $m^u_t[k] += sp^u_t[k]; \quad m^l_t[k] += sp^l_t[k]$
        //temporal update (Equation 6.6)
        if $t < T'$ then
            $m^u_{t+1}[k] = \alpha * z^u_t; \quad m^l_{t+1}[k] = \alpha * z^l_t;$
        end if
        //fire (Equation 6.4)
        $s^u_t[k] = fire(m^u_t[k] - th_f); \quad s^l_t[k] = fire(m^l_t[k] - th_f);$
    end for
end for
Return upper and lower bound of intermediate data $\dot{x}^u_t$, $\dot{x}^l_t$, $m^u_t$, $m^l_t$, $s^u_t$, $s^l_t$
```

Algorithm 7: S-CROWN

```
Input:
$\dot{x}^u_t$, $\dot{x}^l_t$, $m^u_t$, $m^l_t$, $s^u_t$, $s^l_t$ from S-IBP;     robust training time steps $T'$;
Func:
//S-CROWN
build identity $I$ matrix;     $As_t[K] = I/T'$;
for $k = K$ to 1 do
  initial $Am_t[k] = 0$ for all time steps
  for $t = T'$ to 1 do
    //fire (Equation 6.5)
    build $m_t[k] * d1^l + b1^l \leq s_t[k] \leq m_t[k] * d1^u + b1^u$;
    $Am_t[k] \mathrel{+}= As_t[k]^- * d1^u + As_t[k]^+ * d1^l$;
    $bias \quad \mathrel{+}= As_t[k]^- * b1^u + As_t[k]^+ * b1^l$;
    //temporal update (Equation 6.7)
    if $t > 1$ then
      build $m_{t-1}[k] * d2^l + (1 - s_{t-1}[k]) * d3^l \leq z_{t-1}[k]$;
      build $m_{t-1}[k] * d2^u + (1 - s_{t-1}[k]) * d3^u \geq z_{t-1}[k]$;
      $Am_{t-1}[k] = \alpha * (Am_t^-[k] * d2^u + Am_t^+[k] * d2^l)$;
      $tmp_s \quad = \alpha * (Am_t^-[k] * d3^u + Am_t^+[k] * d3^l)$;
      $As_{t-1}[k] \mathrel{-}= tmp_s$;     $bias \mathrel{+}= sum(tmp_s)$;
    end if
    //spatial update;
    $As_t[k-1] = Am_t[k] * w[k-1]$;     $bias \mathrel{+}= Am_t[k] * b[k-1]$;
  end for
end for
Return $\sum_t \big(As_t[0] * (\dot{x}^u_t + \dot{x}^l_t)/2 - |As_t[0]| * (\dot{x}^u_t - \dot{x}^l_t)/2\big) + bias$
```

and q_t represent the input and output of non-linear function $g(\cdot)$. Based on our analysis of the $fire$ function and temporal update, we can bound each non-linear operation with linear upper and lower bounds, which can be represented as

$$p_t * d^l + b^l \leq g(p_t) \leq p_t * d^u + b^u. \tag{6.13}$$

Here d and b represent the slope and intercept of a linear function. Then, the lower bound of the output can be formulated as

$$\begin{aligned} f(\dot{x}) &\geq \textstyle\sum_t Aq_t * g(p_t) + B \\ &\geq \textstyle\sum_t \underbrace{(Aq_t^- * d^u + Aq_t^+ * d^l)}_{Ap_t} \ * \ p_t \ + \\ &\quad \underbrace{(Aq_t^- * b^u + Aq_t^+ * b^l + B)}_{new\ bias\ B}. \end{aligned} \tag{6.14}$$

We use A^+ and A^- to represent the positive and negative values in matrix A. Equation 6.14 can be used to process the $fire$ and temporal update in SNN. For the linear operation, i.e., $q_t = w * p_t + b$, the boundary propagation can be formulated as

$$\begin{aligned} f(\dot{x}) &\geq \textstyle\sum_t Aq_t * (w * p_t + b) + B \\ &\geq \textstyle\sum_t \underbrace{(Aq_t * w)}_{Ap_t} \ * \ p_t \ + \underbrace{(Aq_t * b + B)}_{new\ bias\ B}. \end{aligned} \tag{6.15}$$

Now, we can follow the computation process provided in Algorithm 7 to compute the slope matrix and bias of the bounded spike input ($\dot{x}^u$ and $\dot{x}^l$). Finally, we can follow the robust training method provided in [46] and use the lower bound of S-CROWN to train the SNN model.

6.3 Evaluation

6.3.1 Experiment Setup

Dataset and Network Structure: In this work, we evaluate our robust training method on three datasets: MNIST [79], FMNIST [117] and NMNIST [82]. MNIST and

FMNIST are digital datasets and NMNIST is spike dataset. For each dataset, we use two network structures in experiments, i.e., a three-layers FC network and a four-layers CONV network. The detailed setting for datasets and network structures are shown in Table 6.3.1. We set the firing threshold th_f and decay factor α to 0.25 for the neuron modeling in Equation 2.1 and 2.2.

	MNIST	FMNIST	NMNIST
Input Type	digital	digital	spike
Size	1*28*28	1*28*28	2*34*34
Time Step	10	10	10
FC Acc	98.45%	87.58%	98.30%
CONV Acc	99.09%	89.53%	99.05%
FC	X-FC512-FC256-FC10		
CONV	X-C64K3S2-C128K3S2-FC256-FC10		

Table 6.1: Datasets and network structure

Original and Robust Training: In original training, we adopt BPTT based training method [20]. We train 80 epochs for each SNN model. During the original training, the learning rate is set to 0.01 at the beginning, it decays to 0.001 at the 55^{th} epoch. In robust training, we use the lower bound of S-CROWN as the loss function. During robust training, we set ϵ to 0 at the beginning. It will increase linearly to the final ϵ during the first 250 training epochs. In the last 50 training epochs, ϵ is unchanged.

Adversarial Attack: In this work we adopt the untargeted white-box gradient-based attack in SNN, which is introduced in Chapter 5. In our experiment, we select 300 examples for each dataset to apply adversarial attack. In order to bound the noise of adversarial examples, we involve additional constraints during the attack. Specifically, for digital inputs, we clip the adversarial example for each attack iteration to make the adversarial example stay in the boundary. For spike inputs, each attack iteration we limit the amount of changed elements to $size(\hat{x}) * \epsilon/2$ and $size(\hat{x}) * \epsilon/6$ for FC and CONV networks to achieve the highest attack efficiency.

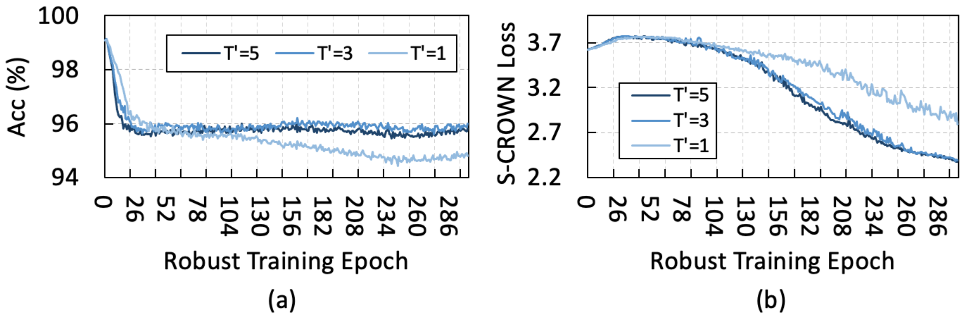


Figure 6.4: Impact of robust training time steps T' on (a) original accuracy and (b) S-CROWN loss. (CONV model; MNIST dataset; $\epsilon = 0.12$)

6.3.2 Robust Training in SNNs

Selection of Robust Training Time Steps T'

We do not follow the original time steps during the robust training. Because the spatial computations in SNNs share the same parameters between different time steps, we can use arbitrary robust training time steps T'. In Figure 6.4 we analyzed the impact of T' on original accuracy and S-CROWN loss. Note that we would like to keep a higher original accuracy but reduce the S-CROWN loss after the robust training. From the result, we can find that $T' = 3$ gives the optimal solution. It implies that a too smaller T' cannot capture the temporal dynamics of SNNs and a larger T' may cause the boundary functions in S-CROWN to become too loose. Also we find that the robust training time for each epoch is 2.6×, 7.7×, and 12.9× with respect to the original plain training when we set T' to 1, 3, and 5. Thus, the selection of T' also influences the robust training efficiency. In the rest of our evaluations, we set $T' = 3$ for all robust training.

Robust Training on Variaous Datasets

: The analysis of robust training on different datasets with various ϵ and network structures is shown in Figure 6.5. From the result, we have the following observations:

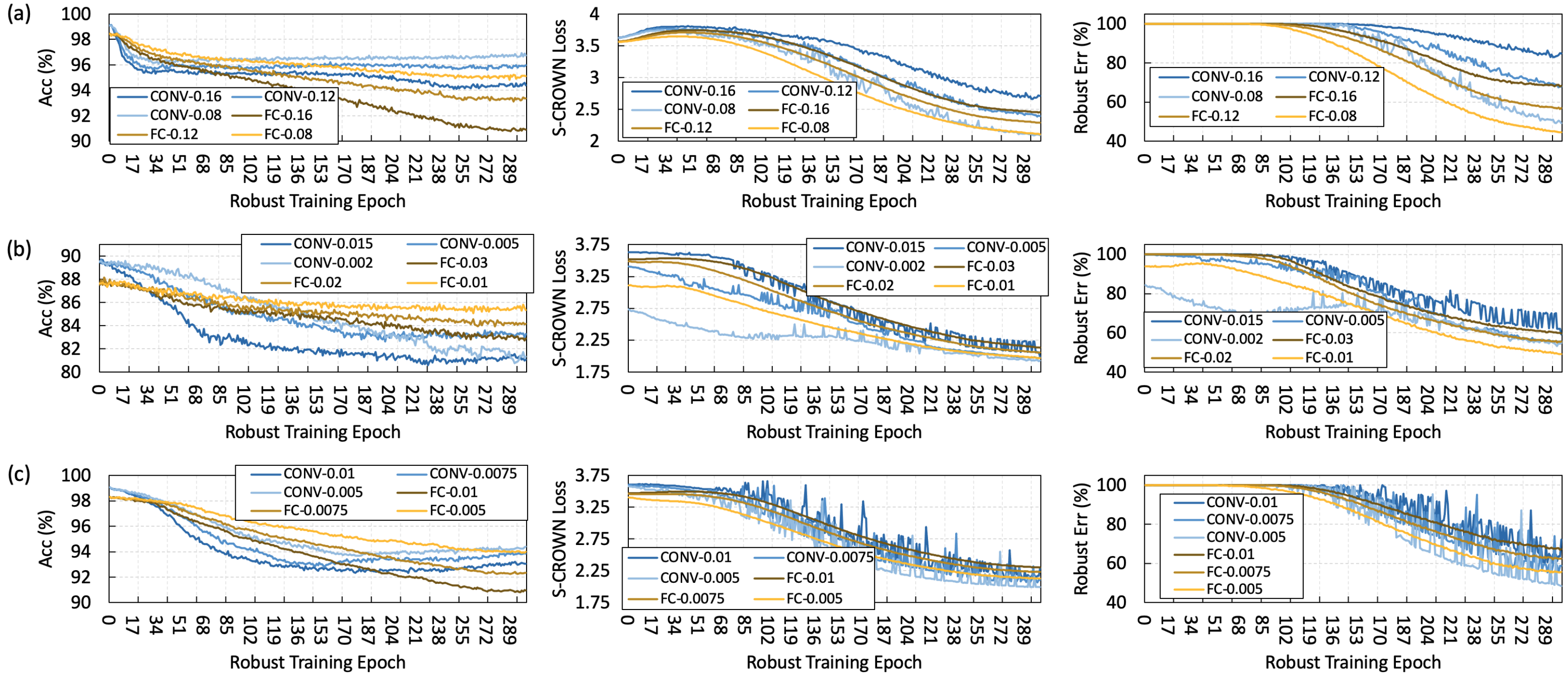


Figure 6.5: Original accuracy, S-CROWN loss and S-CROWN robust error during robust training under (a) MNIST; (b) FMNIST and (C) NMNIST datasets with different network structures and ϵ.

1. During the robust training, it is more stable for a fully connected network(yellow curves are smoother than blue curves) because of the simpler network structure. 2. With a larger ϵ, the original accuracy is dropped more after the robust training. Also, it is more difficult to achieve a smaller robust error with larger ϵ. 3. In the MNIST dataset, when the network structure is CONV, the robust training is more stable as ϵ becomes larger. The potential reason is that when ϵ is small, the flipped regions in spike inputs are more diverse. 4. In FMNIST and NMNIST datasets, when the network structure is CONV, the robust training is much more fluctuating. For FMNIST dataset, the unstable may be caused by the more complicated input data (cloths) and the lower convergence in original training (final accuracy is 89.53%). For NMNIST dataset, the unstable may come from the larger input data size. A larger input size indicates the potential input regions that can be attacked becomes more. By considering the original accuracy and S-CROWN result after the robust training, we select the ϵ as shown in Table 6.3.2. We use FC-ϵ and CONV-ϵ to represent the noise boundary we selected for different network structures during robust training.

Network Robustness Evaluation

We use untargeted white-box adversarial attack to compare the robustness between the original model and the model after robust training. The robustness comparison is shown in Table 6.3.2. We use the original error rate to reflect the accuracy of a model on the 300 test data. The original error may have variance during the evaluation for the digital image dataset because of the input sampling mechanism. In our experiment, we select the attack ϵ to achieve approximately 20%, 30%, 40%, and 50% attack error rate on the original model. From the result, we can find that after the robust training, the models are harder to be attacked with adversarial example in all cases. We also find that the model after robust training is more secure when the attack ϵ is larger, even though

	MNIST (FC-ϵ=0.12, CONV-ϵ=0.12)					FMNIST (FC-ϵ=0.01, CONV-ϵ=0.005)					NMNIST (FC-ϵ=0.005, CONV-ϵ=0.005)				
	attack	original network		robust network		attack	original network		robust network		attack	original network		robust network	
	ϵ	org err	attack err	org err	attack err	ϵ	org err	attack err	org err	attack err	ϵ	org err	attack err	org err	attack err
FC	0.104	0.3%	19.3%	4.7%	16.6% (-2.7%)	0.040	13.0%	20.7%	13.6%	20.0%(-0.7%)	0.0007	3.7%	21.0%	3.7%	10.0%(-11.0%)
	0.124	0.3%	29.7%	4.7%	20.3% (-9.4%)	0.070	13.0%	29.7%	13.3%	24.0%(-5.7%)	0.0009	3.7%	30.7%	3.7%	12.3%(-18.4%)
	0.140	0.3%	39.0%	4.7%	23.3%(-15.7%)	0.100	11.7%	41.3%	14.0%	32.0%(-9.3%)	0.0012	3.7%	41.3%	3.7%	16.7%(-24.6%)
	0.154	0.3%	50.0%	5.0%	24.3%(-25.7%)	0.114	13.3%	49.7%	13.6%	38.7%(-11.0%)	0.0014	3.7%	48.7%	3.7%	18.3%(-30.4%)
CONV	0.120	0.3%	18.7%	3.7%	8.3%(-10.4%)	0.040	10.0%	19.7%	13.7%	22.0%(+0.3%)	0.0008	1.3%	20.0%	4.3%	7.0%(-13.0%)
	0.140	0.3%	29.0%	4.0%	9.0%(-20.0%)	0.060	10.3%	28.3%	17.3%	23.0%(-5.3%)	0.0010	1.3%	33.7%	4.3%	11.0%(-22.7%)
	0.170	0.3%	40.7%	3.3%	10.7%(-30.0%)	0.080	10.0%	41.3%	17.0%	25.7%(-15.6%)	0.0012	1.3%	41.0%	4.3%	11.0%(-30.0%)
	0.190	0.3%	50.7%	4.0%	13.0%(-37.7%)	0.100	10.3%	52.3%	16.7%	34.3%(-18.0%)	0.0015	1.3%	49.0%	4.3%	12.0%(-37.0%)

Table 6.2: Comparing untargeted white-box gradient-based attack between the original model and the model after robustness training.

the ϵ for robust training is far smaller. The potential reason is that the binary behavior of the spike events causes the boundary propagation to diverge quickly, which makes the final boundary cover more noisy inputs. Finally, we find that model robustness can be improved more when the network structure is CONV, since more paramters can be adjusted under the CONV model. From the result, we find that the CONV model in MNIST achieves the highest robustness improvement, i.e. the attack error rate reduces 37.7% with 3.7% original accuracy loss when the attack ϵ equals 0.190.

6.4 Conclusion

For ANNs, training a neural network with certified methods not only shows remarkable efficiency to improve the model's robustness but also presents flexibility to cooperate with other robust training methods. In this Chapter, we aim to design an efficient robust training method for SNNs based on the certified methods. Specifically, we design S-IBP and S-CROWN to tackle the distinct non-linear neuron behaviors in SNNs. Also, we formulate the input boundary for different input types. We evaluate the models' robustness to untargeted white-box adversarial attack. Based on our results, we can achieve at most 37.7% attack error reduction with 3.7% original accuracy loss, which demonstrates the efficiency of our proposed method.

Chapter 7

Summary & Future Work

7.1 Summary

Recently, extensive studies on spiking neural networks (SNNs) are motivated by bio-plausible neuron modeling, based on the observations that neurons use spike signals to represent information and communicate with each other. How to train an SNN model with expected functionality is an essential topic. In the early stage, the bio-inspired unsupervised learning methods are designed to train an SNN model, such as STDP. However, these training methods always suffer low accuracy issues. In order to improve the accuracy of SNNs, the BPTT-based learning method is designed that makes SNNs can achieve considerable accuracy for general tasks.

However, the BPTT-based learning method is not optimized by the mainstream platform for neural network training such as GPUs. In this dissertation, we first propose an accelerator *H2Learn* to optimize different training stages in BPTT-based training according to their characteristics. Specifically, we design LUT-based architecture to utilize the binary format of the spike during the Forward Pass and Weight Update phases. In the Backward Pass, we design sparse-aware architecture to efficiently reduce the computation overhead for sparse input and output.

In the meantime, we also want to investigate whether the BPTT-based SNN training efficiency can be improved on GPUs. Based on our observations, we find that the temporal update and batch-normalization occupy non-trivial memory and computation resources. According to the observations, we redesign the dataflow during the training. Specifically, we only store the result after convolution and matrix multiplication in the forward pass to save the memory consumption. The abandoned intermediate data will be recomputed in the weight update and backward pass phases. In order to accelerate the computation, we design kernels to fuse temporal update and batch normalization. With the optimized kernels, the kernel launching time is reduced significantly.

Besides improving the BPTT-based training efficiency, this dissertation also explores the security issue for SNNs. In the beginning, we investigate the adversarial attack on SNNs. We design a framework to generate adversarial examples for both spike and digital inputs. Our framework also includes techniques to reduce the noise in the adversarial examples and handle the gradient vanishing problem during the adversarial example generation. Based on our evaluation, we find that the gradient-based attack may fall into a ‘trap’ region because of the rate coding on the output neurons. We further adjust the firing threshold to make the adversarial attack in SNNs more effective.

At last, we propose a robust training method for SNNs against adversarial examples. Certified defense is one of the most efficient methods to train a robust neural network model. However, existing methods cannot be directly applied to the SNNs. Firstly, the spike inputs or Bernoulli-based sampling on digital inputs require new input boundary formulations. Secondly, the temporal update behavior is different from the existing non-linear functions in ANNs. Thus, we formalize ℓ_0-norm and ℓ_∞-norm boundaries for spike and digital inputs, respectively. Also, we design S-IBP and S-CROWN to tackle the non-linear fire function and temporal update in SNNs which firstly introduce SNNs to the linear relaxation-based verification family.

In conclusion, this dissertation focuses on providing an efficient and robust neuromorphic system. We hope this dissertation would help to enhance the architecture design for the neuromorphic chips, and inspire the algorithm development for the security-aware applications when adopting the neuromorphic system.

7.2 Future Work

Recently, SNNs achieve considerable accuracy improvement in general tasks. However, many research fields of SNNs are still not deeply explored. Here we list several potential directions for future work on SNNs.

Application & Network Model: Currently, for most applications, the accuracy of SNNs is still hard to surpass the accuracy of ANNs such as Transformers. Thus, it is urgent to find a more suitable task for SNNs especially after considering the energy consumption during the inference. In general, SNNs have the potential to handle applications with continuous temporal activities. Instead of focusing on the application, it is also interesting to explore how to apply the characteristics of SNNs to other NN models such as Graph Neural Networks (GNNs) and Transfromers.

Neuromorphic Chips: Since SNNs show distinguishable characteristics compared to ANNs, it is promising to design neuromorphic chips to make the SNN inference and training more efficient. Furthermore, building an integrated neuromorphic chip is another interesting direction. Specifically, SNNs have different coding mechanisms (i.e., rate coding, temporal coding, phase coding, etc), neuron modeling (LIF model, IF model, etc), and learning methods (BPTT, STDP, ANN-SNN conversion, etc). It is meaningful to build a neuromorphic chip that can support multiple SNN features and reuse most of the hardware resources.

Secure Neuromorphic System: A secure neuromorphic system includes many aspects. Firstly, we need to improve the robustness of an SNN against adversarial attacks. Here, we suggest analyzing the SNN robustness with different SNN features like coding mechanisms, neuron modeling, and learning methods, even for the different input formats. Instead of considering the adversarial attack, how to protect an SNN model to be stoled by an attacker is important, since the SNN model is the intellectual property of the model provider. Also, the spike inputs from the client-side should be protected at the same time, since the inputs may expose the client's privacy, especially for biological applications.

Bibliography

[1] S. Ghosh-Dastidar and H. Adeli, *Spiking neural networks*, *International journal of neural systems* **19** (2009), no. 04 295–308.

[2] W. Maass, *Networks of spiking neurons: the third generation of neural network models*, *Neural networks* **10** (1997), no. 9 1659–1671.

[3] L. Deng, Y. Wu, X. Hu, L. Liang, Y. Ding, G. Li, G. Zhao, P. Li, and Y. Xie, *Rethinking the performance comparison between snns and anns*, *Neural Networks* **121** (2020) 294–307.

[4] W. Maass, *Noise as a resource for computation and learning in networks of spiking neurons*, *Proceedings of the IEEE* **102** (2014), no. 5 860–880.

[5] G. Haessig, A. Cassidy, R. Alvarez, R. Benosman, and G. Orchard, *Spiking optical flow for event-based sensors using ibm's truenorth neurosynaptic system*, .

[6] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, *Direct training for spiking neural networks: Faster, larger, better*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1311–1318, 2019.

[7] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza, *Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high speed scenarios*, *IEEE Robotics & Automation Letters* **3** (2018), no. 2 994–1001.

[8] Z. Jonke, S. Habenschuss, and W. Maass, *Solving constraint satisfaction problems with networks of spiking neurons*, *Frontiers in neuroscience* **10** (2016) 118.

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

[10] G. Shi, Z. Liu, X. Wang, C. T. Li, and X. Gu, *Object-dependent sparse representation for extracellular spike detection*, *Neurocomputing* **266** (2017) 674–686.

[11] T. Hwu, J. Isbell, N. Oros, and J. Krichmar, *A self-driving robot using deep convolutional neural networks on neuromorphic hardware*, in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 635–641, IEEE, 2017.

[12] S. Song, K. D. Miller, and L. F. Abbott, *Competitive hebbian learning through spike-timing-dependent synaptic plasticity*, *Nature neuroscience* **3** (2000), no. 9 919–926.

[13] P. U. Diehl and M. Cook, *Unsupervised learning of digit recognition using spike-timing-dependent plasticity*, *Frontiers in computational neuroscience* **9** (2015) 99.

[14] A. Tavanaei and A. S. Maida, *Bio-inspired spiking convolutional neural network using layer-wise sparse coding and stdp learning*, .

[15] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, *Stdp-based spiking deep neural networks for object recognition*, *Neural Networks the Official Journal of the International Neural Network Society* **99** (2016) 56.

[16] J. Liu, H. Huo, W. Hu, and T. Fang, *Brain-inspired hierarchical spiking neural network using unsupervised stdp rule for image classification*, .

[17] C. Lee, G. Srinivasan, P. Panda, and K. Roy, *Deep spiking convolutional neural network trained with unsupervised spike-timing-dependent plasticity*, *IEEE Transactions on Cognitive and Developmental Systems* **11** (2019), no. 3 384–394.

[18] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, *Stdp-based spiking deep convolutional neural networks for object recognition*, *Neural Networks* **99** (2018) 56–67.

[19] J. H. Lee, T. Delbruck, and M. Pfeiffer, *Training deep spiking neural networks using backpropagation*, *Frontiers in neuroscience* **10** (2016) 508.

[20] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, *Spatio-temporal backpropagation for training high-performance spiking neural networks*, *Frontiers in neuroscience* **12** (2018).

[21] Y. Jin, W. Zhang, and P. Li, *Hybrid macro/micro level backpropagation for training deep spiking neural networks*, in *Advances in neural information processing systems*, pp. 7005–7015, 2018.

[22] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, *Long short-term memory and learning-to-learn in networks of spiking neurons*, in *Advances in Neural Information Processing Systems*, pp. 787–797, 2018.

[23] S. B. Shrestha and G. Orchard, *Slayer: Spike layer error reassignment in time*, .

[24] P. Gu, R. Xiao, G. Pan, and H. Tang, *Stca: Spatio-temporal credit assignment with delayed feedback in deep spiking neural networks.*, in *IJCAI*, pp. 1366–1372, 2019.

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

[26] X. Jin, A. Rast, F. Galluppi, S. Davies, and S. Furber, *Implementing spike-timing-dependent plasticity on spinnaker neuromorphic hardware*, in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2010.

[27] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, and G. Indiveri, *A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses*, *Frontiers in neuroscience* **9** (2015) 141.

[28] C. Frenkel, M. Lefebvre, J.-D. Legat, and D. Bol, *A 0.086-mm^2 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm cmos*, *IEEE transactions on biomedical circuits and systems* **13** (2018), no. 1 145–158.

[29] C. Frenkel, J.-D. Legat, and D. Bol, *Morphic: A 65-nm 738k-synapse/mm^2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning*, *IEEE Transactions on Biomedical Circuits and Systems* **13** (2019), no. 5 999–1010.

[30] E. Baek, H. Lee, Y. Kim, and J. Kim, *Flexlearn: fast and highly efficient brain simulations using flexible on-chip learning*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 304–318, 2019.

[31] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, *Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations*, *Proceedings of the IEEE* **102** (2014), no. 5 699–716.

[32] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, *et. al.*, *A million spiking-neuron integrated circuit with a scalable communication network and interface*, *Science* **345** (2014), no. 6197 668–673.

[33] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, *A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic*

asynchronous processors (dynaps), *IEEE transactions on biomedical circuits and systems* **12** (2017), no. 1 106–122.

[34] J. Pei, L. Deng, S. Song, M. Zhao, Y. Zhang, S. Wu, G. Wang, Z. Zou, Z. Wu, W. He, *et. al.*, *Towards artificial general intelligence with hybrid tianjic chip architecture*, *Nature* **572** (2019), no. 7767 106–111.

[35] L. Deng, G. Wang, G. Li, S. Li, L. Liang, M. Zhu, Y. Wu, Z. Yang, Z. Zou, J. Pei, *et. al.*, *Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation*, *IEEE Journal of Solid-State Circuits* (2020).

[36] S. Narayanan, K. Taht, R. Balasubramonian, E. Giacomin, and P.-E. Gaillardon, *Spinalflow: An architecture and dataflow tailored for spiking neural networks*, .

[37] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing properties of neural networks*, *arXiv preprint arXiv:1312.6199* (2013).

[38] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, *arXiv preprint arXiv:1412.6572* (2014).

[39] A. Kurakin, I. Goodfellow, and S. Bengio, *Adversarial examples in the physical world*, *arXiv preprint arXiv:1607.02533* (2016).

[40] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, *Deepfool: a simple and accurate method to fool deep neural networks*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

[41] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, *The limitations of deep learning in adversarial settings*, in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 372–387, IEEE, 2016.

[42] N. Carlini and D. Wagner, *Towards evaluating the robustness of neural networks*, in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 39–57, IEEE, 2017.

[43] W. Brendel, J. Rauber, and M. Bethge, *Decision-based adversarial attacks: Reliable attacks against black-box machine learning models*, *arXiv preprint arXiv:1712.04248* (2017).

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

[45] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. Mann, and P. Kohli, *On the effectiveness of interval bound propagation for training verifiably robust models*, *arXiv preprint arXiv:1810.12715* (2018).

[46] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh, *Towards stable and efficient training of verifiably robust neural networks*, in *International Conference on Learning Representations*, 2020.

[47] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh, *Automatic perturbation analysis for scalable certified robustness and beyond*, *Advances in Neural Information Processing Systems* **33** (2020).

[48] L. Liang, Z. Qu, Z. Chen, F. Tu, Y. Wu, L. Deng, G. Li, P. Li, and Y. Xie, *H2learn: High-efficiency learning accelerator for high-accuracy spiking neural networks*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[49] L. Ling, C. Zhaodong, D. Lei, T. Fengbin, L. Guoqi, and X. Yuan, *Accelerating spatiotemporal supervised training of large-scale spiking neural networks on gpu*, in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022.

[50] L. Liang, X. Hu, L. Deng, Y. Wu, G. Li, Y. Ding, P. Li, and Y. Xie, *Exploring adversarial attack in spiking neural networks with spike-compatible gradient*, *IEEE transactions on neural networks and learning systems* (2021).

[51] L. Liang, K. Xu, X. Hu, L. Deng, and Y. Xie, *Toward robust spiking neural network against adversarial perturbation*, *arXiv preprint arXiv:2205.01625* (2022).

[52] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition.* Cambridge University Press, 2014.

[53] A. Grüning and S. M. Bohte, *Spiking neural networks: Principles and challenges.*, in *ESANN*, Citeseer, 2014.

[54] J. Vreeken *et. al.*, *Spiking neural networks, an introduction*, .

[55] P. Falez, P. Tirilly, and I. M. Bilasco, *Improving stdp-based visual feature learning with whitening*, in *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2020.

[56] P. Ferré, F. Mamalet, and S. J. Thorpe, *Unsupervised feature learning with winner-takes-all based stdp*, *Frontiers in computational neuroscience* **12** (2018) 24.

[57] G. Srinivasan and K. Roy, *Restocnet: Residual stochastic binary convolutional spiking neural network for memory-efficient neuromorphic computing*, *Frontiers in neuroscience* **13** (2019) 189.

[58] S. Oh, Y. Shi, J. Del Valle, P. Salev, Y. Lu, Z. Huang, Y. Kalcheim, I. K. Schuller, and D. Kuzum, *Energy-efficient mott activation neuron for full-hardware implementation of neural networks*, *Nature Nanotechnology* **16** (2021), no. 6 680–687.

[59] A. Roy, S. Venkataramani, N. Gala, S. Sen, K. Veezhinathan, and A. Raghunathan, *A programmable event-driven architecture for evaluating spiking neural networks*, in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2017.

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

[61] M. Heidarpur, A. Ahmadi, M. Ahmadi, and M. R. Azghadi, *Cordic-snn: On-fpga stdp learning with izhikevich neurons*, *IEEE Transactions on Circuits and Systems I: Regular Papers* **66** (2019), no. 7 2651–2661.

[62] B. Glackin, T. M. McGinnity, L. P. Maguire, Q. Wu, and A. Belatreche, *A novel approach for the implementation of large scale spiking neural networks on fpga hardware*, in *International Work-Conference on Artificial Neural Networks*, pp. 552–563, Springer, 2005.

[63] S. R. Kulkarni, S. Yin, J.-s. Seo, and B. Rajendran, *An on-chip learning accelerator for spiking neural networks using stt-ram crossbar arrays*, in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1019–1024, IEEE, 2020.

[64] A. K. Fidjeland and M. P. Shanahan, *Accelerated simulation of spiking neural networks using gpus*, in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2010.

[65] Y. Yu and N. K. Jha, *Spring: A sparsity-aware reduced-precision monolithic 3d cnn accelerator architecture for training and inference*, *IEEE Transactions on Emerging Topics in Computing* (2020).

[66] H. Nakahara, Y. Sada, M. Shimoda, K. Sayama, A. Jinguji, and S. Sato, *Fpga-based training accelerator utilizing sparseness of convolutional neural network*, in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 180–186, IEEE, 2019.

[67] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, *Multiscale co-design analysis of energy, latency, area, and accuracy of a reram analog neural training accelerator*, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **8** (2018), no. 1 86–101.

[68] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, J. Yan, and X. Li, *Tnpu: An efficient accelerator architecture for training convolutional neural networks*, in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 450–455, 2019.

[69] J. Büchel, G. Lenz, Y. Hu, S. Sheik, and M. Sorbaro, *Adversarial attacks on spiking convolutional networks for event-based vision*, *arXiv preprint arXiv:2110.02929* (2021).

[70] S. Sharmin, P. Panda, S. S. Sarwar, C. Lee, W. Ponghiran, and K. Roy, *A comprehensive analysis on adversarial robustness of spiking neural networks*, in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[71] A. Bagheri, O. Simeone, and B. Rajendran, *Adversarial training for probabilistic spiking neural networks*, in *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 1–5, IEEE, 2018.

[72] A. Marchisio, G. Pira, M. Martina, G. Masera, and M. Shafique, *Dvs-attacks: Adversarial attacks on dynamic vision sensors for spiking neural networks*, in *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–9, IEEE, 2021.

[73] R. El-Allami, A. Marchisio, M. Shafique, and I. Alouani, *Securing deep spiking neural networks against adversarial attacks through inherent structural parameters*, in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 774–779, IEEE, 2021.

[74] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, *Cusparse library*, in *GPU Technology Conference*, 2010.

[75] T. P. Lillicrap and A. Santoro, *Backpropagation through time and the brain*, *Current opinion in neurobiology* **55** (2019) 82–89.

[76] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, *Backpropagation and the brain*, *Nature Reviews Neuroscience* (2020) 1–12.

[77] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, *Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, IEEE, 2020.

[78] F. Tu, W. Wu, Y. Wang, H. Chen, F. Xiong, M. Shi, N. Li, J. Deng, T. Chen, L. Liu, *et. al.*, *Evolver: A deep learning processor with on-device quantization-voltage-frequency tuning*, *IEEE Journal of Solid-State Circuits* (2020).

[79] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, *Proceedings of the IEEE* **86** (1998), no. 11 2278–2324.

[80] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, tech. rep., Citeseer, 2009.

[81] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[82] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, *Converting static image datasets to spiking neuromorphic datasets using saccades*, *Frontiers in neuroscience* **9** (2015) 437.

[83] H. Li, H. Liu, X. Ji, G. Li, and L. Shi, *Cifar10-dvs: An event-stream dataset for object classification*, *Frontiers in neuroscience* **11** (2017) 309.

[84] P. Lichtsteiner, C. Posch, and T. Delbruck, *A 128× 128 120 db 15 μs latency asynchronous temporal contrast vision sensor*, *IEEE Journal of Solid-State Circuits* **43** (2008), no. 2 566–576.

[85] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, *Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques*, in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 694–701, IEEE, 2011.

[86] T. NVIDIA, *V100 gpu architecture whitepaper*, .

[87] X. Ju, B. Fang, R. Yan, X. Xu, and H. Tang, *An fpga implementation of deep spiking neural networks for low-power and fast classification*, *Neural computation* **32** (2020), no. 1 182–204.

[88] K. Stewart, G. Orchard, S. B. Shrestha, and E. Neftci, *On-chip few-shot learning with surrogate gradient descent on a neuromorphic processor*, in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 223–227, IEEE, 2020.

[89] S. B. Shrestha, "Slayer for loihi." `https://flagship.kip.uni-heidelberg.de/jss/HBPm?m=displayPresentation&mI=209&mEID=8359`.

[90] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, *Going deeper in spiking neural networks: Vgg and residual architectures*, *Frontiers in neuroscience* **13** (2019) 95.

[91] N. Rathi and K. Roy, *Diet-snn: A low-latency spiking neural network with direct input encoding and leakage and threshold optimization*, *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[92] N. Rathi, G. Srinivasan, P. Panda, and K. Roy, *Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation*, *arXiv preprint arXiv:2005.01807* (2020).

[93] G. Srinivasan, C. Lee, A. Sengupta, P. Panda, S. S. Sarwar, and K. Roy, *Training deep spiking neural networks for energy-efficient neuromorphic computing*, in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8549–8553, IEEE, 2020.

[94] X. Xie, H. Qu, G. Liu, M. Zhang, and J. Kurths, *An efficient supervised training algorithm for multilayer spiking neural networks*, *PloS one* **11** (2016), no. 4 e0150329.

[95] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, *Going deeper with directly-trained larger spiking neural networks*, *arXiv preprint arXiv:2011.05280* (2020).

[96] J. Pu, V. P. Nambiar, A. T. Do, and W. L. Goh, *Block-based spiking neural network hardware with deme genetic algorithm*, in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2019.

[97] A. Agrawal, A. Ankit, and K. Roy, *Spare: Spiking neural network acceleration using rom-embedded rams as in-memory-computation primitives*, *IEEE Transactions on Computers* **68** (2018), no. 8 1190–1200.

[98] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie, *fusegnn: accelerating graph convolutional neural network training on gpgpu*, in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[99] J. Fang, Y. Yu, C. Zhao, and J. Zhou, *Turbotransformers: An efficient gpu serving system for transformer models*, *arXiv preprint arXiv:2010.05680* (2020).

[100] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, *Going deeper with directly-trained larger spiking neural networks*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 11062–11070, 2021.

[101] PyTorchContributors., *Torch script. https://pytorch.org/docs/ master/jit.html., Accessed: 2018-09-24.* (2018).

[102] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer, *Adversarial patch*, *arXiv preprint arXiv:1712.09665* (2017).

[103] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, *Robust physical-world attacks on deep learning models*, *arXiv preprint arXiv:1707.08945* (2017).

[104] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, *Trojaning attack on neural networks, in 25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221* (2018).

[105] K. Pei, Y. Cao, J. Yang, and S. Jana, *Deepxplore: Automated whitebox testing of deep learning systems*, in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18, ACM, 2017.

[106] Y. X. M. Tan, Y. Elovici, and A. Binder, *Exploring the back alleys: Analysing the robustness of alternative neural network architectures against adversarial attacks*, *arXiv preprint arXiv:1912.03609* (2019).

[107] A. Marchisio, G. Nanfa, F. Khalid, M. A. Hanif, M. Martina, and M. Shafique, *Snn under attack: are spiking deep belief networks vulnerable to adversarial examples?*, *arXiv preprint arXiv:1902.01147* (2019).

[108] I. M. Comsa, T. Fischbacher, K. Potempa, A. Gesmundo, L. Versari, and J. Alakuijala, *Temporal coding in spiking neural networks with alpha synaptic function*, in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8529–8533, IEEE, 2020.

[109] H. Mostafa, *Supervised learning based on temporal coding in spiking neural networks*, *IEEE transactions on neural networks and learning systems* **29** (2017), no. 7 3227–3235.

[110] S. R. Kheradpisheh and T. Masquelier, *S4nn: temporal backpropagation for spiking neural networks with one spike per neuron*, *arXiv preprint arXiv:1910.09495* (2019).

[111] D. Su, H. Zhang, H. Chen, J. Yi, P.-Y. Chen, and Y. Gao, *Is robustness the cost of accuracy?–a comprehensive study on the robustness of 18 deep image classification models*, in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 631–648, 2018.

[112] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, *et. al.*, *A low power, fully event-based gesture recognition system*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7243–7252, 2017.

[113] A. Athalye, N. Carlini, and D. Wagner, *Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples*, in *International conference on machine learning*, pp. 274–283, PMLR, 2018.

[114] K. Xu, G. Zhang, S. Liu, Q. Fan, M. Sun, H. Chen, P.-Y. Chen, Y. Wang, and X. Lin, *Adversarial t-shirt! evading person detectors in a physical world*, in *European Conference on Computer Vision*, pp. 665–681, Springer, 2020.

[115] E. Wong and J. Z. Kolter, *Provable defenses against adversarial examples via the convex outer adversarial polytope*, in *International Conference on Machine Learning*, 2018.

[116] M. Mirman, T. Gehr, and M. Vechev, *Differentiable abstract interpretation for provably robust neural networks*, in *International Conference on Machine Learning*, pp. 3575–3583, 2018.

[117] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms*, *arXiv preprint arXiv:1708.07747* (2017).

[118] L. Patrick, C. Posch, and T. Delbruck, *A 128x 128 120 db 15μ s latency asynchronous temporal contrast vision sensor*, *IEEE journal of solid-state circuits* **43** (2008) 566–576.

[119] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, *Efficient neural network robustness certification with general activation functions*, *arXiv preprint arXiv:1811.00866* (2018).