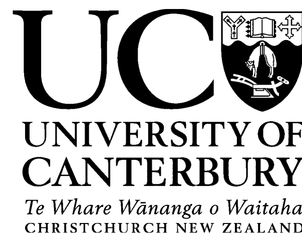
Bio-inspired Spiking Neural Network Algorithm Development and Tactile Signal Processing

Chunming Jiang

Supervised by
Professor Yilei Zhang

A thesis presented for partial fulfilment of the requirements for the degree of Doctor of Philosophy
in
Mechanical Engineering
at the



University of Canterbury
Christchurch, New Zealand
December 2022

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Professor Yilei Zhang. It has been a great honor and pleasure to be one of your students and to work with you. Pursuing a PhD is a long process and is never easy. Your deep confidence in my abilities and work means a lot to me. Thank you for your vast knowledge, unfailing support, enthusiasm, patience, and valuable insights that led me into the study of brain-like neural networks. You have made this journey so interesting and exciting with your endless enthusiasm and energy. Thank you for being a mentor and role model on my path to becoming a researcher. You have been and will always be a mentor in my life and research career. Thank you for all that you have done.

I would also like to express my deepest gratitude to my co-mentor, Professor Le Yang. Thank you for your selfless contribution of your ideas and suggestions. Your insightful advice, never wavering support and patient guidance have been instrumental in this journey. Thank you for being approachable no matter where you are. Thank you for your kind advice on my career. Thank you so much for your encouragement and constructive advice on my research, never throwing even a little cold water on any of the ideas I presented. Without your guidance and care, I would never have reached this stage.

Special thanks to the University of Canterbury for providing a free, relaxed and positive learning and living environment, to the Department of Mechanical Engineering for providing me with study and living expenses, allowing me more time to focus on my own research, and to the faculty and technicians of the Department of Mechanical Engineering for their help. I would like to thank Research Compute Cluster for providing me with computing resources to support my high performance computing tasks.

I would like to thank my parents most of all for your unconditional love and support throughout the years. When I needed support, you were always there. To my friends, thank you for being there for me when I was lost. You guys mean a lot to me. Finally, I want to thank New Zealand and the city of Christchurch, I love New Zealand and I love Christchurch!

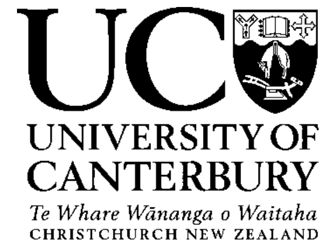
Abstract

Spiking neural networks (SNNs) are a new generation of deep learning models inspired by biology, which belong to a subset of deep learning and have a strong biological basis to support them. It has received more and more attention from researchers in recent years due to its advantages of high efficiency, energy saving, and high interpretability. However, compared with traditional ANNs, SNNs are still in the early stage of research and still face many problems.

In this thesis, we first analyze the reasons of the poor performance of SNNs in image classification and propose a new interpretable spiking neuron to improve the learning ability of the network for big datasets. In addition, we propose a new method of adversarial defense to enhance the robustness of SNNs against tiny noise. Besides, we also propose a new training algorithm for optimizing the speed of the SNN in the training and inference process.

In addition to the study of SNN algorithms, we also apply SNN to specific application problems. To address the problem of redundancy in event camera datasets, we propose a SNN-based mask network that selectively deletes redundant pulse signals, thus reducing the space occupied by the dataset and facilitating transmission. Finally, we combine SNNs with engineering problems, and since SNNs are good at handling timing signals, we use SNNs to achieve high-precision classification of tactile signals collected by tactile sensors.

In summary, SNNs are positioned as ANNs with biological plausibility, i.e., they have the interpretability of biological networks and some characteristics of ANNs. In this thesis, we work on developing different algorithms for the speed, accuracy, and robustness of SNNs and design SNNs for problems under a variety of different domains.



Deputy Vice-Chancellor's Office
Postgraduate Research Office

Publications

This form is to accompany the submission of any thesis that contains research reported in co-authored work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from the extract comes:

Chapter 2

- **Chunming Jiang,** Yilei Zhang, "KLIF: An optimized spiking neuron unit for tuning surrogate gradient slope and membrane potential", IEEE transactions on neural networks and learning systems, IEEE transactions on neural networks and learning systems, submitted in May, 2022 **[Under Review]**

Chapter 3

- **Chunming Jiang,** Yilei Zhang, " Adversarial Defense via Neural Oscillation inspired Gradient Masking", IEEE transactions on neural networks and learning systems, submitted in Oct, 2022, https://arxiv.org/abs/2211.02223 **[Under Review]**

Chapter 4

- **Chunming Jiang,** Yilei Zhang,"A noise based novel strategy for faster SNN training", Neural Computation, submitted in Oct, 2022, https://arxiv.org/abs/2211.05453 **[Under Review]**

Chapter 5

- **Chunming Jiang,** Yilei Zhang, " Spiking sampling network for sparse representation", Neurocomputing, submitted in Aug, 2022, https://arxiv.org/abs/2211.04166 **[Under Review]**

Chapter 6

- **Chunming Jiang,** Le Yang, Yilei Zhang, "A Spiking Neural Network With Spike-Timing-Dependent Plasticity for Surface Roughness Analysis", IEEE Sensors Journal (Volume: 22, Issue: 1, 01 January 2022)

Please detail the nature and extent (%) of contribution by the candidate:

The contribution of the candidate to the publication amounts to ~95% based on the overall time and effort dedicated to the final outcome. The candidate developed, implemented, and led all research work. All analytical work was developed and performed by the candidate in consultation with the supervisors.

Certification by Co-authors:

If there is more than one co-author then a single co-author can sign on behalf of all

The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Doctoral candidate's contribution to this co-authored work
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text

Name: Yilei Zhang Signature: Yilei Zhang Date: 30/12/2022

Contents

Acknowledgement i

Abstract iii

Publications iv

Contents i

List of Figures vi

List of Tables vi

1 Introduction x

1.1 Spiking neural network and its challenge 1

1.2 Noise role in biological neural system 3

1.3 Noise role in artificial neural networks 6

1.4 Preface 8

2 KLIF: An optimized spiking neuron unit for tuning surrogate gradient slope and membrane potential 11

2.1 Introduction 12

2.2 Method 15

2.2.1 k-based leaky Integrate-and-Fire model 15

2.2.2 Adaptive surrogate gradient descent 17

2.2.3 Activation function ReLU 20

2.2.4 Encoding and decoding schemes 21

2.3 Results and discussions 22

2.3.1 Comparison of LIF and KLIF 23

2.3.2 Ablation study 25

2.3.3 Biological plausibility of KLIF 27

2.4 Conclusion 29

3 Adversarial Defense via Neural Oscillation inspired Gradient Masking 30

3.1 Introduction 31

3.2 Preliminaries 33

3.2.1 SNNs and biological neural oscillation 33

3.2.2 Adversarial attacks 35

3.3 Experiments 37

3.3.1 Datasets and Models 37

3.3.2 Neural oscillation neuron 38

3.3.3 Alternative Neural oscillation neuron 39

3.3.4 Adversarial defense strategy 40

3.4 Results 42

3.4.1 Robustness analysis of neural oscillation model 42

3.4.2 Robustness analysis of alternative neural oscillation model 43

3.4.3 Validation of defense 46

3.5 Discussion and conclusion 48

3.6 Supplementary 49

3.6.1 Parameter values for reproducibility 49

3.6.2 Neuron performance testing 49

3.6.3 Function selection of alternative neural oscillation 51
3.6.4 Firing property of neural oscillation neuron 52
4 A noise based novel strategy for faster SNN training 53
4.1 Introduction 54
4.2 Methods 56
4.2.1 Leaky Integrate-and-Fire model 56
4.2.2 Training single-step SNN and converting to multi-step SNN 57
4.3 Results and discussions 60
4.3.1 Inference accuracy 61
4.3.2 Comparison of training and inference time with related work 62
4.3.3 The impact of α 64
4.3.4 The impact of noise type 65
4.3.5 Biological plausibility of uniform noise distribution in neuron 66
4.4 Discussion and conclusion 68
5 Spiking sampling network for image sparse representation and dynamic vision sensor data compression 69
5.1 Introduction 70
5.2 Methods 72
5.2.1 Leaky Integrate-and-Fire (LIF) model 72
5.2.2 Architecture and training of spiking sampling network 73
5.2.3 Data compression of dynamic vision sensor 76
5.3 Experiments 77
5.3.1 Image reconstruction comparison 77

5.3.2 Event data compression 79

5.3.3 Specificity and universality 81

5.4 Discussion and conclusion 83

5.5 Supplementary 83

6 A Spiking Neural Network with Spike-timing-dependent Plasticity for Surface Roughness Analysis 85

6.1 Introduction 86

6.2 Methods 88

6.2.1 Experimental Setup 88

6.2.2 SNN architecture 89

6.2.3 Neuron and synapse model 90

6.2.4 Dataset 94

6.2.5 Input Encoding 95

6.2.6 Training and classification 97

6.3 Results 97

6.4 Conclusion 102

6.5 Supplementary 102

Chapter 7 Conclusions 104

Chapter 8 Discussion and future Work 108

References 110

APPENDIX A 130

APPENDIX B 139

APPENDIX C 147

APPENDIX D 148

APPENDIX E 150

List of Figures

Figure 1.1 Flowchart of the thesis organization..9

Figure 2.1 Structure of spiking neurons. (a) leaky Integrate-and-Fire (LIF) model. (b) k-based leaky Integrate-and-Fire (KLIF) model. The dotted box represents the dynamic of function Ft. It incorporates a scaling factor k and an activation function $ReLU$. 15

Figure 2.2 Derivative approximation of the non-differentiable spike activity. (a) step activation function of the spike activity and its original derivative function which is infinite value at H = 1 and zero value at other points. (b) scaling factor k to adjust the slope of the surrogate gradient curve $\partial S \partial F$. 18

Figure 2.3 Encoder and decoder of SNNs..21

Figure 2.4 The test accuracy of KLIF v.s. LIF neurons on different datasets during training. 23

Figure 2.5 The change of scaling factor ki in the i-th layer during training on **a.** CIFAR-10 and **b.** DVS128-Gesture..26

Figure 2.6 The distribution of firing rate for neurons in each layer during training on CIFAR-10...26

Figure 3.1 The training process of the model with alternative neural oscillation neurons at time t. The model trained first with neural oscillation neurons can be regarded as a 'teacher model'. It provides the labels for a 'student model' which replaces neural oscillation neurons with alternative neural oscillation neurons. The 'student model' keeps the same trained weights and fits spike trains $\boldsymbol{S}'jt$ of student model to $\boldsymbol{S}jt$ of teacher model by learning variables a and b in each layer. 38

Figure 3.2 (a). The solid and dotted orange lines represent $\partial S \partial H$ of neural oscillation model when γ is -0.2 and 0.8, respectively. The red line is $\partial S' \partial H$ of alternative neural oscillation

model. (b). Partial enlargement of graph (a) in the green dashed circle. 41

Figure 3.3 Accuracy on (a)VGG-16/CIFAR-10 (b)ResNet-18/CIFAR-10 when using neural oscillation neuron (blue curve) and alternative neural oscillation neuron (orange curve) 50

Figure 3.4 (a) Curve of function F to fit the noise item on VGG-16/CIFAR-10. (b) Gradient curve $\partial S' t \partial H t$ when using different F. 50

Figure 3.5 Spontaneous spike firing of neural oscillation neuron. 51

Figure 4.1 Three steps of our method to train a SNN model. Step 1, single-step SNN training with noise distribution $Nnoise$. Step 2, copy N single-step SNNs and ensemble them together. $Nnoise$ varies over time-step t. Step 3, establish the temporal correlation among N different models. 56

Figure 4.2 Inference accuracy of models on different datasets with T = 1, 5, 10 while training with $Nnoise$. 61

Figure 4.3 Training speed of SNNs when directly training an SNN(T=10) by the surrogate gradient approach versus training a 10-step SNN by our approach. 62

Figure 4.4 Training time of SNNs when directly training an SNN(T=10) by the surrogate gradient approach versus training a a 10-step SNN by our approach. The benchmark inference accuracy of the three models is 92%, 88%, and 90%, respectively. 63

Figure 4.5 The impact of α on CIFAR10/VGG-16 and CIFAR10/ResNet-18. The black dotted line represents the accuracy of trained single-step SNN. 64

Figure 4.6 Inference accuracy of models on different datasets with T = 1, 10 while training using Gaussian noise and uniform noise, respectively. 66

Figure 4.7 (a) Neural potential dynamic in the absence of input. (b) Neural potential dynamic when receiving input. (c) Subthreshold membrane potential oscillation. Source: Figure (c) is cited from [136]. 67

Figure 5.1 Architecture of the spiking sampling network. The output of spiking sampling

network is a mask of the same size as the input. 72

Figure 5.2 Comparison between random sampling and spiking sampling on (a) MNIST and (b) CIFAR-10. (c) Sampled pixels by spiking sampling network on CIFAR-10. 75

Figure 5.3 Different sampling rate comparison of random sampling and spiking sampling on MNIST 76

Figure 5.4 (a) Compression of N-MNIST dataset. (b) Classification validation of compressed N-MNIST dataset. 76

Figure 5.5 (a) Test classification accuracy on N-MNIST with different sampling method and rate. (b) data size comparison after different compressing rate by spiking sampling. The numbers on the bars represent the average number of spikes retained per sample for the dataset. 79

Figure 5.6 Reconstruction comparison of 10% random sampling and 10% spiking sampling on the main reconstruction network trained by random sampling. 81

Figure 5.7 Reconstruction comparison of 10% random sampling and 10% spiking sampling on the main reconstruction network trained by spiking sampling. 82

Figure 6.1 (a) The structure of designed biomimetic artificial fingertip. (b) Biomimetic fingertip sliding along the test surface. (c) Eight solid nickel test surfaces with different roughness values. 88

Figure 6.2 Illustration of SNN network structure for tactile signal processing. 89

Figure 6.3 (a) LIF neuron model. (b) Schematic of the classic STDP. 90

Figure 6.4 Typical tactile signals generated by two perpendicular PVDF films when sliding on eight surfaces with different roughness values. PVDF1 is perpendicular to the sliding direction, while PVDF2 is parallel to the sliding direction. 94

Figure 6.5 The split and combination of raw data. 95

Figure 6.6 An original static image is encoded into a spike map over various time steps using

rate coding..96

Figure 6.7 (a) Test accuracy comparison between augmented PVDF1 and augmented PVDF2 datasets with 15 training epochs. (b) Test accuracy comparison between augmented PVDF1 dataset and original PVDF1 dataset. (c) Test accuracy comparison among the different numbers of excitatory neurons in the output layer. (d) Test accuracy comparison among fusing both augmented PVDF data and single augmented PVDF data with 15 training epochs................98

Figure 6.8 (a) Standard deviation features from the tactile signals of the two PVDF films when sliding on eight surfaces with different surface roughness values. (b) Sum of absolute values from the tactile signals of the two PVDF films when sliding on eight surfaces with different surface roughness values...99

List of Tables

Table 2.1 Network structures and training details for different datasets. 22

Table 2.2 Comparison between our work and the state-of-the-art methods on different datasets. 24

Table 2.3 Ablation Study of KLIF.on CIFAR-10 27

Table 2.4 Accuracy of using KLIF/KLIF* 28

Table 2.5 Accuracy of using KLIF with different activation functions on CIFAR-10 28

Table 3.1 Top-1 Accuracy (%) on clean images using two kinds of oscillation neurons 44

Table 3.2 Neuron model summary under different attack scenarios......................................44

Table 3.3 Top-1 classification accuracy (%) under the scenario 1 attack..............................44

Table 3.4 Top-1 classification accuracy (%) under the scenario 2 attack..............................44

Table 3.5 Top-1 classification accuracy (%) under the scenario 3 attack..............................45

Table 3.6 Top-1 classification accuracy (%) under the scenario 4 attack..............................45

Table 3.7 Top-1 classification accuracy (%) under the scenario 5 attack..............................45

Table 3.8 White-box robustness (accuracy (%)) on CIFAR-10 using the ResNet-18 (ϵ=8/255) 45

Table 3.9 Noise range [a,b] and values of c and d of alternative neural oscillation model....49

Table 3.10 Top-1 classification accuracy (%) under the scenario 4 attack............................51

Table 4.1 Network structures and training epoch for different datasets. 61

Table 4.2 Inference time comparison between our work and related work 64

Table 5.1 Network structures for image reconstruction on MNIST and CIFAR-10 and data compression on N-MNIST...84

Table 6.1 Classifier: SNN with 400 output neurons 101

Table 6.2 The highest classification accuracy of different methods 101

Table 6.3 Parameters in SNN 103

1 Introduction

1.1 Spiking neural network and its challenge

Spiking neural networks (SNNs) are artificial neural networks that more closely mimic biological neural networks. In addition to neurons and synaptic states, SNNs incorporate the concept of time into their operational model. The idea is that instead of transmitting information at each propagation cycle (as in a typical multilayer perceptron network), neurons in SNNs transmit information when the membrane potential reaches a specific value, called the threshold. When the membrane potential reaches threshold, the neuron activates and generates a signal that is transmitted to other neurons, which in turn increase or decrease their potential in response to this signal.

One of challenges of SNNs is how to train and optimize the parameters. Currently, the existing training methods of SNNs can be classified into three types: (1) unsupervised learning (2) indirect supervised learning (3) direct supervised learning. The first one is inspired by the weight modification of synapses between biological neurons. For example, spike time-dependent plasticity (STDP) [1-3]. Since it relies mainly on local neuronal activity rather than global supervision, STDP-based unsupervised algorithms have so far been limited to shallow SNNs, yielding accuracies significantly lower than those provided by ANN on complex datasets such as CIFAR-10 [4-6]. The second approach is to first train an ANN and then convert it to a SNN version with the same network structure, where the rate of the SNN neurons can be approximated as the analog output of the ANN neurons. ANN-to-SNN conversion has achieved the state-of-the-art (SOTA) SNNs for image recognition tasks that performs close or even better than the conventional ANNs [7], however, the inference time of SNNs converted from ANNs still requires a lot of time (about several thousand time steps) and memory, leading to increased

latency as well as decreased energy efficiency, which diminish the benefits of spiking [7-9]. In addition, ANN-to-SNN is only suitable to static datasets currently, not neuromorphic datasets. The last one is the direct supervised learning, which adopts mainly the same gradient descent algorithm with ANNs. Spikeprop pioneered the gradient descent method to train multilayer SNNs [10, 11]. It assumes that each neuron spikes once in a given time period to encode the input signal, and minimizes the difference between the network output and the desired signal by calculating the gradient associated with these firing times. As a result, the result is low latency. Despite these advantages, the use of only a single spike per neuron has its limitations and is less suitable for processing temporal stimuli such as electroencephalography (EEG) signals, speech or video [12]. Other subsequent works like Tempotron [13], ReSuMe [14], and SPAN [15] can emit multiple spikes, but they can only be applied to single-layered networks. An approach treated the membrane potential as a differentiable signal to solve the problem of non-differentiation of spikes and proposed a straightforward BP algorithm to train deep SNNs with multiple spikes [16]. Recently, Wu et al. proposed a spatiotemporal back-propagation training framework for SNNs, which introduces a differentiable surrogate function to approximate the derivative of spike activity [17]. This method combines both the spatial domain and the temporal domain in the training phase and has yielded best results for deep convolutional SNNs in small-scale image recognition datasets such as digit classification on the MNIST. However, for large-scale tasks, it has not been able to outperform the conversion-based approach or ANNs in terms of accuracy [7]. In addition, since SNNs introduce a temporal dimension, direct training of SNNs often takes several times more training time than training an ANN, which seriously consumes computational resources.

To improve the performance of SNNs and decrease gaps between ANNs and SNNs, Wu et al. proposed neuron normalization [17]. This mechanism can balance the firing rate of each neuron

and avoid the loss of important information. Cheng et al. added the lateral connections between neighboring neurons and get better accuracy [18]. Some researchers have revised the neuron model's parameters to improve the accuracy. For example, the learnable membrane time constants in Leaky Integrate-and-Fire (LIF) neurons were utilized to make the charging and leakage process more flexible [19, 20], and an adaptive threshold spiking neuron model was proposed to enhance learning capabilities of SNNs [21].

In summary, research on SNNs is still in its early stages. In this thesis, we research and develop the algorithms for the following main problems faced by SNNs:

1. SNN performance remains low compared with the corresponding ANN architecture.
2. The training speed of SNNs is slow.
3. SNNs are vulnerable to network attacks and noise and lack specific defense approaches.

1.2 Noise role in biological neural system

Noise sources also widely exist in the brain. It is well known that neurons can react differently, even when the same stimuli are given [22, 23]. They are comprised of intrinsic sources such as open-close fluctuation of membrane resistance and extrinsic source triggered mainly by signal transmission and network effects [24].

External noise

When an organism receives external stimuli, these external sensory stimuli are generally noisy, and these noises come from thermodynamic or quantum mechanical properties. For instance, thermodynamic noise affects all types of chemical sensing (including smell and gustation)

because molecules diffuse into the receptor at random rates and because receptor proteins have a limited capacity to precisely count the amount of signalling molecules [25, 26]. In a manner similar to vision, photons that enter the photoreceptor at a rate determined by a Poisson process must be absorbed. This sets a physical cap on contrast sensitivity in vision, which is diminished under low light conditions when fewer photons reach the photoreceptor [27].

Cellular noise

Neuronal trial-to-trial variability is influenced by a number of variables. Changes in the internal states of neurons and neural networks, as well as random events inside neurons and neural networks, are examples of these [28, 29]. Each neuron has noise that builds up as a result of randomness in the cellular machinery that processes information [23]; this noise can then get greater as a result of nonlinear calculations and network interactions. Numerous stochastic processes operate at the biochemical and biophysical levels in neurons.

Electrical noise and action potentials (APs)

Both carrying APs and performing local computations employ the membrane potential. The mechanisms underlying variations in resting membrane potential [30, 31] (membrane-potential fluctuations) and AP threshold [32] have only lately come to light, despite the fact that these variables have long been researched. Even in the absence of synaptic inputs, electrical noise in neurones results in variations in membrane potential. Channel noise [33-35], which is created when voltage- or ligand-gated ion channels randomly open and close, is the main source of this electrical noise. The fluctuation of the AP threshold at nodes of Ranvier [36] and the dependability of AP initiation in membrane patches can both be explained by channel noise,

according to stochastic models [37, 38]. Furthermore, patch-clamp experiments in vitro demonstrate that membrane-potential fluctuations caused by channel noise in the dendrites and the soma are significant enough to impact AP timing [39-42]. Channel noise can have an impact on the start and spread of APs.

Synaptic noise

Another significant source of brain noise is synapses. Gradient potentials in the postsynaptic neurone cause a spontaneous action potential when neurotransmitter-containing vesicles randomly exocytose and bind to the postsynaptic membrane [43]. It is regarded as the source of noise with the highest amplitude in the cerebral cortex [44].

Benefits of noise

Although there exist so many random noise sources in neurons and synapses, experiments prove that noise is essential in neural activities and plays a vital role in processes such as decision-making, signal detection, and memory [45], and the nervous system can respond accurately and reliably and shows good robustness under different levels of noise [46-48]. In addition, the neural network formed in the presence of noise will be more robust and explore more states, which will help learn and adapt to the changing dynamic environment [49, 50]. For instance, stochastic resonance is a mechanism by which the presence of a certain amount of noise can improve the capability of threshold-like systems to receive and transmit weak (periodic) signals [51, 52]. Few signals are detected at low noise levels because the sensory signal does not push the system over the threshold. The response is dominated by the noise at high noise levels. However, for intermediate noise strengths, the noise permits the signal to

cross the threshold without obstructing it. Numerous sensory systems have shown stochastic resonance-type effects since they were originally discovered in the visual neurones of the cat [53]. These include crayfish mechanoreceptors [54], shark multimodal sensory cells [55], cricket sensory neurones [55], and human muscle spindles [56]. Both passively electrically induced paddlefish [57] and human balance control [58] have been used to directly illustrate and regulate the behavioural consequences of stochastic resonance. Additionally, sub-threshold inputs have little impact on the system's output in spike-generating neurones. Such threshold nonlinearities can be transformed by noise, which increases the likelihood that sub-threshold inputs will exceed the threshold the closer the inputs are to the threshold. As a result, when averaged over time, this noise effectively creates a smoothed nonlinear [37]. According to research on contrast invariance of orientation tuning in the primary visual cortex [59], this makes spike initiation easier and can enhance neural-network behaviour. Additionally, neuronal networks that have grown up in a noisy environment will be more resilient and explore more states, which will aid in learning and adaptation to the shifting requirements of a dynamic environment [50, 60].

1.3 Noise role in artificial neural networks

Noise is not only widespread in biological neural systems, but in current artificial neural networks, a large number of cases have been implemented by introducing noise and randomization into neural nets to achieve a wide variety of learning tasks.

Generalization

When you start studying neural networks, one of the first things you learn is what overfitting and underfitting are. When you train a neural network with a tiny dataset, the network typically memorises the training dataset rather than learning generic aspects of our data, therefore it can be difficult to create a model that perfectly generalises your data. This is especially true when you have a little dataset. Because of this, the model will perform well with training data but poorly with fresh data (for instance: the test dataset). A tiny dataset offers a poor description of our problem, making it challenging to learn from. Getting more data requires a lot of effort. However, there are situations when you can use certain procedures to improve the performance of your model.

An important role of noise in artificial neural networks is to generalize the network and prevent overfitting. Data augmentation [61] improves the network's ability to generalize the data and enhances network robustness by making random changes to the input, for example, rotating, stretching, or adding random noise to the input image in computer vision. In addition to adding noise to the input, noise can be added in multiple locations, such as in weights, labels, or a separate network layer. In [62, 63], noise is applied to neural network weights rather than hidden layers. It is also possible to think of stochastic ensemble learning [64] and learning with stochastic depth [65] as noise injection methods for weights or architecture. Dropout [66] is another operation that introduces randomization. It randomly cuts the connections between neurons during the training process to avoid overfitting the network to the data.

Robustness

Another important application area of noise is in the robustness of networks. Adversarial attacks [67-69] attack the network by adding imperceptible noise to the input image, prompting

the network to produce incorrect outputs. For this type of adversarial attacks, researchers have also proposed randomization-based defenses. For example, [70] adds a random noise layer to the network to disrupt adversarial attacks. [71-74] add samples with noise in the training process for adversarial training, thus improving the resistance of the network to attacks.

Imgae generation

Noise also has a wide and important application in the field of image generation. For example, the variational autoencoder [75], when the latent space is obtained from the encoder input, enables the decoder to sample the latent space by introducing noise, and thus generates more reasonable generation samples. Like generative adversarial network [76], random Gaussian noise is used as input to generate specific objects through the generator. The recently popular diffusion model [77] can be used for a variety of image generation tasks by gradually adding Gaussian noise while gradually learning and removing it through the neural network.

1.4 Preface

In this chapter, we introduce the background of spiking neural network and its facing challenges currently, and we also present the presence and role of noise in the biological nervous system and its application in artificial neural networks. The following chapters of this thesis are broadly structured as Figure 1.1 shows.

Chapter 2-4 focus on the algorithms’ development of SNN:

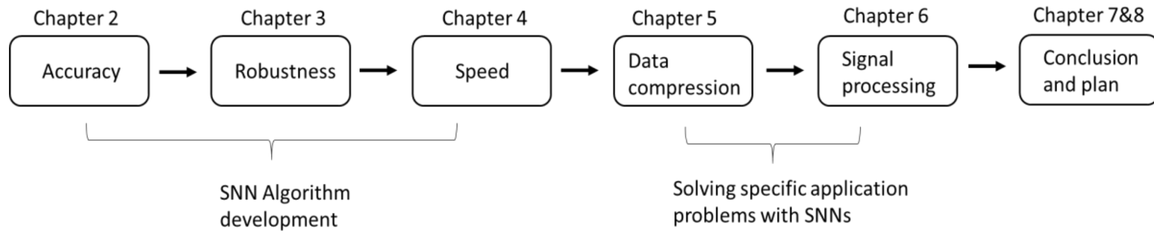


Figure 1.1 Flowchart of the thesis organization

- Chapter 2 develops a bio-inspired neural model dynamically adjusting the surrogate gradient curve in spiking neural networks. The proposed neural model can greatly improve classification accuracy on different visual datasets.
- Chapter 3 presents a neural oscillation model-based approach integrating noise distribution to improve the robustness of SNNs and defend against adversarial attacks. Proposed neural models are robust to various gradient-based adversarial attacks.
- Chapter 4 introduces a noise-based ensemble learning algorithm to accelerate the training and inference of SNN. The approach introduces noise distribution to replace membrane potential during training. Compared to the previous training methods, ours can reduce training time by 65%-75% and achieves more than 100 times faster inference speed.

Chapter 5-6 apply SNNs on different specific problems:

- Chapter 5 presents the application of a two-layer biological plausible SNN on processing tactile signals. Collected tactile signals of surface roughness by a bionic tactile sensor and recognized them by a biologically plausible unsupervised neural network. Proposed a data augmentation approach to learn surface features in the case of a few samples.

- Chapter 6 develops a data-dependent sampling and reconstruction network consisting of a spiking neural network for adaptive sampling and another network for data reconstruction. Tested the proposed network using event camera data and verify its data compression capability.

Chapters 7 and 8 provides the summary for this thesis and discussed about the future work for research complement.

2 KLIF: An optimized spiking neuron unit for tuning surrogate gradient slope and membrane potential

Abstract

Spiking neural networks (SNNs) have attracted much attention due to their ability to process temporal information, low power consumption, and higher biological plausibility. However, it is still challenging to develop efficient and high-performing learning algorithms for SNNs. Methods like artificial neural network (ANN)-to-SNN conversion can transform ANNs to SNNs with slight performance loss, but it needs a long simulation to approximate the rate coding. Directly training SNN by spike-based backpropagation (BP) such as surrogate gradient approximation is more flexible. Yet now, the performance of SNNs is not competitive compared with ANNs. In this chapter, we propose a novel k-based leaky Integrate-and-Fire (KLIF) neuron model to improve the learning ability of SNNs. Compared with the popular leaky integrate-and-fire (LIF) model, KLIF adds a learnable scaling factor to dynamically update the slope and width of the surrogate gradient curve during training and incorporates a ReLU activation function that selectively delivers membrane potential to spike firing and resetting. The proposed spiking unit is evaluated on both static MNIST, Fashion-MNIST, CIFAR-10 datasets, as well as neuromorphic N-MNIST, CIFAR10-DVS, and DVS128-Gesture datasets. Experiments indicate that KLIF performs much better than LIF without introducing additional computational cost and achieves state-of-the-art performance on these datasets with few time steps. Also, KLIF is believed to be more biological plausible than LIF. The good performance of KLIF can make it completely replace the role of LIF in SNN for various tasks.

2.1 Introduction

Artificial neural networks (ANNs) have achieved remarkable success in many domains in recent years. Record accuracy at tasks such as image recognition [78-80], image segmentation [81], and language translation [82] has been achieved. However, their success is highly dependent on high-precision digital conversion [8], which requires large amounts of energy and memory. Therefore, deploying conventional ANNs on embedded platforms with limited energy and memory is still challenging.

Spiking neural networks (SNNs), regarded as the third generation of neural networks, were inspired by the biological neural system, and they mimic how information is transmitted in the human brain [83]. Unlike conventional ANNs, spiking neurons communicate and compute through discrete-time sparse events rather than continuous values. Due to being event-driven, SNNs are more efficient in terms of energy and memory consumption on embedded platforms. So far, SNNs have been used for kinds of tasks, such as image[84] and voice recognition [85].

One of the challenges in SNNs is how to train and optimize the network parameters. Currently, the existing training methods of SNNs can be classified into three types: (1) unsupervised learning, (2) indirect supervised learning, (3) direct supervised learning. The first one is inspired by the weight modification of synapses between biological neurons. A classic example is the spike time-dependent plasticity (STDP) [1-3]. Since it relies mainly on local neuronal activity rather than global supervision, STDP-based unsupervised algorithms have been limited to shallow SNNs with ≤5 layers, yielding accuracy significantly lower than those provided by ANNs on complex datasets such as CIFAR-10 [4-6].

The second approach is to train an ANN model firstly and then convert it to SNN with the same network structure, where the firing rate of the SNN neurons can be approximated as the analog output of the ANN neurons. For image recognition tasks, ANN-to-SNN conversion has led to state-of-the-art (SOTA) SNNs that perform close or even better than the conventional ANNs [7]. However, SNNs converted from ANNs still require a lot of inference time (about several thousand time steps) and a large amount of memory, leading to increased latency and decreased energy efficiency, which diminishes the benefits of spiking models [7-9]. The last SNN training technique is direct supervised learning, which adopts mainly the same gradient descent algorithm as in ANNs. Spikeprop pioneered the gradient descent method to train multilayer SNNs [10, 11]. It assumes that each neuron fires once in a given time window to encode the input signal and minimizes the difference between the network output and desired signal by calculating the gradient associated with these firing times.

Nevertheless, the use of only a single spike per neuron has its limitations and is less suitable for processing temporal stimuli such as electroencephalography (EEG) signals, speech, or video [12]. Other subsequent works like Tempotron [13], ReSuMe[14], and SPAN [15] can utilize multiple spikes, but they can only be applied to single-layer networks. An approach treated the membrane potential as a differentiable signal to solve the problem of non-differentiation of spikes and proposed a straightforward BP algorithm to train deep SNNs with multiple spikes [16]. Recently, Wu et al. proposed a spatiotemporal backpropagation training framework for SNNs, introducing a differentiable surrogate function to approximate the derivative of spiking activity [17, 86, 87]. This method combines the spatial and temporal domains in the training phase and has yielded the best results for deep convolutional SNNs in small-scale image recognition datasets such as digit classification on the MNIST. However, for

large-scale tasks, it has not been able to outperform the conversion-based approach or ANNs in terms of accuracy [7].

To further improve the performance of SNNs and decrease gaps between ANNs and SNNs, Wu et al. proposed neuron normalization. This mechanism can balance the firing rate of each neuron and avoid the loss of important information. Cheng et al. added the lateral connections between neighboring neurons and obtained better accuracy [18]. Some researchers have revised the neuron model's parameters to improve the accuracy. For example, the learnable membrane time constants in Leaky Integrate-and-Fire (LIF) neurons were utilized to make the charging and leakage process more flexible [19, 20], and an adaptive threshold spiking neuron model was proposed to enhance the learning capabilities of SNNs [21].

In this chapter, we propose a novel spiking neural unit KLIF to replace the commonly adopted LIF model in SNNs. KLIF adds a learnable scaling factor that dynamically updates the slope and width of the surrogate gradient curve during training and accelerates the convergence. It also incorporates a ReLU activation function that selectively delivers membrane potential to spike firing and resetting. We verified our model on both classic static MNIST, Fashion-MNIST, CIFAR-10 datasets widely used in ANNs as benchmarks and neuromorphic N-MNIST, CIFAR10-DVS, DVS128-Gesture datasets. Experiments show that SNN with KLIF improves the test accuracy on all datasets and outperforms the SOTA accuracy.

In summary, our main work is:

1. We propose a novel spiking neural unit KLIF which can improve accuracy of models on different visual tasks by adaptive surrogate gradient descent and potential rectification.

2. We independently analyse the impact of the learnable scaling factor and rectified function. Our experiments show that the scaling factor and the ReLU activation function can independently contribute to improving accuracy of models.

3. We improve the coding layers of SNNs, which contributes to convergence and accuracy improvement of models.

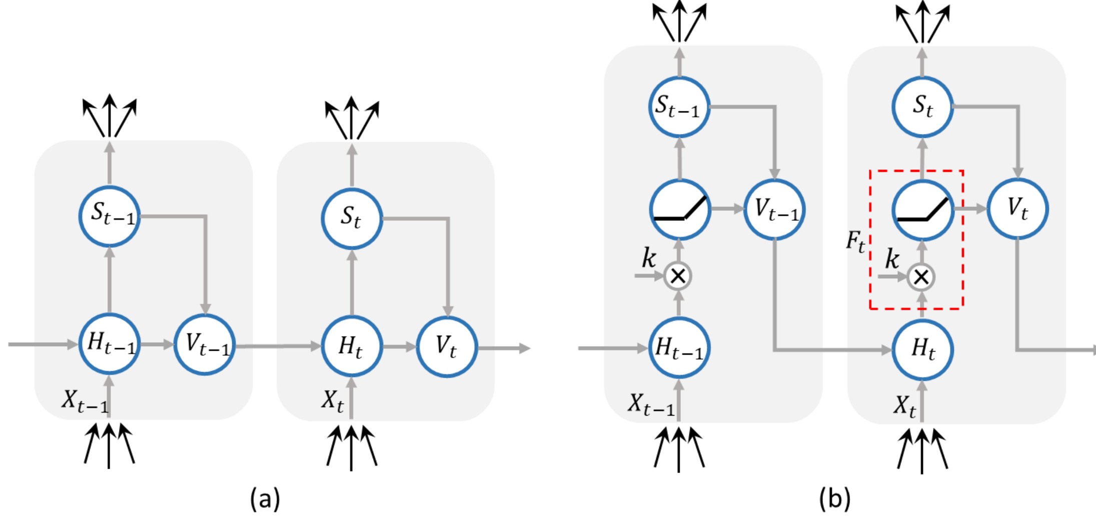


Figure 2.1 Structure of spiking neurons. (a) leaky Integrate-and-Fire (LIF) model. (b) k-based leaky Integrate-and-Fire (KLIF) model. The dotted box represents the dynamic of function F_t. It incorporates a scaling factor k and an activation function $ReLU$.

2.2 Method

In Sec. 2.2.1, we first briefly review the LIF model and then give the dynamic equations of KLIF. In Sec. 2.2.2 and Sec. 2.2.2-2.2.3, we explain the benefits that KLIF brings. Finally, network structures and coding layers used in SNN models, including encoding and decoding, are clarified in Sec. 2.2.4.

2.2.1 k-based leaky Integrate-and-Fire model

The LIF model is one of the fundamental computing units of SNNs. It is a simplified model of biological neurons and describes the non-linear relationship of input and output. The sub-threshold dynamics of the LIF spiking neuron can be modeled using Equation (2.1).

$$\tau \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{\mathrm{reset}}) + X(t) \tag{2.1}$$

Where $V(t)$ is the membrane potential of the neuron and τ is the time constant, $X(t)$ is defined as the weighted sum of the input spikes for each time step.

When a neuron receives inputs from the previous layer, its membrane potential will accumulate. Once the potential value exceeds the neuron's threshold, the neuron will fire a spike and promptly restore the reset potential V_{reset} which is set to be 0 in this paper. To simulate the dynamic actions of LIF neurons discretely in time, we use the difference Equations (2.2)-(2.4) to approximate the continuous dynamic process [19].

$$H_t = V_{t-1} + \frac{1}{\tau}(V_{\mathrm{reset}} - V_{t-1}) + \frac{1}{\tau}X_t \tag{2.2}$$

$$S_t = \begin{cases} 1, & if\ H_t > V_{th} \\ 0, & otherwise \end{cases} \tag{2.3}$$

$$V_t = H_t(1 - S_t) + V_{\mathrm{reset}} S_t = H_t(1 - S_t) \tag{2.4}$$

Where H_t and V_t represent the membrane potentials before and after triggering a spike at time t, respectively. X_t denotes the external input, and τ denotes the time constant with a value of 2. V_{th} denotes the firing threshold, which is 1 in this paper. S_t denotes the output of a neuron

at time t, which equals 1 if there is a spike and 0 otherwise. With Equations (2.2)-(2.4), we describe the charging, firing, and resetting actions of the LIF neuron. Figure 2.1(a) illustrates the dynamics of the LIF neuron.

Unlike the LIF model, we propose the k-based spiking neural unit (KLIF) which adds a function F_t (Equation (2.5)) between charging H_t and firing S_t into the LIF model (Figure 2.1(b)). The function consists of a scaling factor k and an activation function ReLU. As shown in Figure 2.1(b), a spiking neuron accumulate first its potential at time t, then the accumulated potential is multiplied by k and passes through the ReLU function before being compared with the firing threshold. The dynamics of KLIF can be described by Equations (2.2) and (2.5)-(2.7).

$$F_t = \mathrm{ReLU}(kH_t) \tag{2.5}$$

$$S_t = \begin{cases} 1, & if\ F_t > V_{th} \\ 0, & otherwise \end{cases} \tag{2.6}$$

$$V_t = F_t(1 - S_t) + V_{\mathrm{reset}} S_t = F_t(1 - S_t) \tag{2.7}$$

In section 2.3, we will discuss the reason for choosing ReLU. Compared with the LIF model, KLIF brings two benefits: adaptive surrogate gradient descent and membrane potential regulation.

2.2.2 Adaptive surrogate gradient descent

As we all know, ANNs are trained by gradient-based backpropagation (BP), which uses gradient information to optimize the synaptic connections and neuron parameters. Unfortunately, gradient-based optimization fails in SNNs because the firing action is non-differentiable, as described in Equation (2.3). The derivative of S_t is infinite at $H_t = V_{th}$, and the derivative is 0 at other places, as shown in Figure 2.2(a). An approach called surrogate

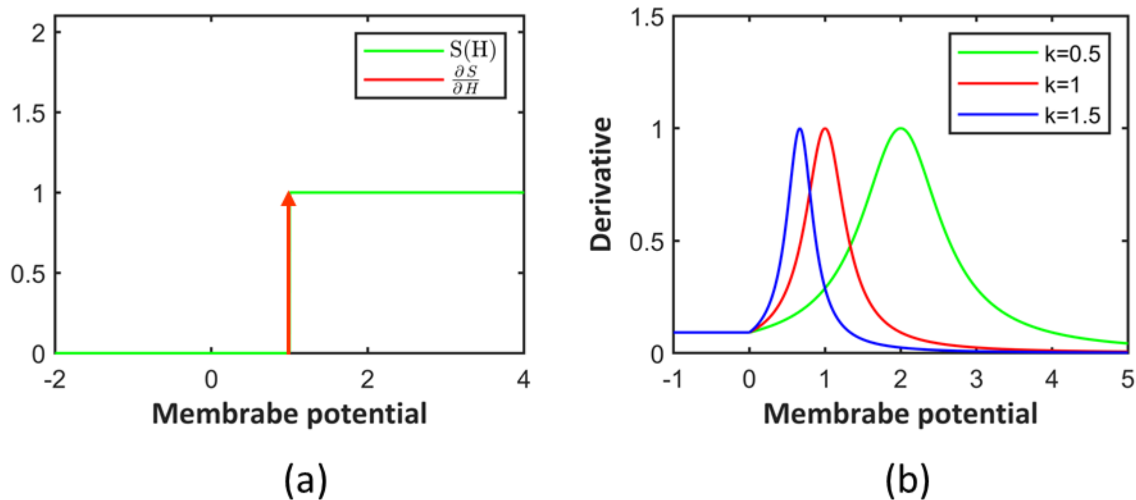


Figure 2.2 Derivative approximation of the non-differentiable spike activity. (a) step activation function of the spike activity and its original derivative function which is infinite value at H = 1 and zero value at other points. (b) scaling factor k to adjust the slope of the surrogate gradient curve $\frac{\partial S}{\partial F}$.

gradient descent was proposed to address this issue [86]. The trick is to replace the derivative of the non-differentiable step function with an approximate differentiable function. It provides surrogate gradients that can be utilized to optimize the parameters of SNN efficiently during backpropagation. The differentiable function could have several forms [88]. The similarity among them is that their primitive function approximates the shape of the step function. Values of the differentiable function are relatively big around the threshold, while those away from the threshold tend to approach 0.

It has been confirmed that the type of curves of the surrogate derivative is not critical to the accuracy and the convergence speed of SNNs, but the proper curve steepness has an impact [86]. The earlier works all set the steepness empirically and do not consider the adjustment of the curve anymore during training [86]. In contrast, our work proposes the learnable scaling factor k which can automatically change during the training process to fit the training data precisely.

The loss function L is defined by the mean squared error (MSE). Under the principle of chain rule [89], we can calculate the gradients of the scaling factor k^i in the i-th layer in the network according to Equation (2.8).

$$\sum_t \frac{\partial L}{\partial k^i} = \sum_t \frac{\partial L}{\partial \boldsymbol{S}_t^i} \frac{\partial \boldsymbol{S}_t^i}{\partial \boldsymbol{F}_t^i} \frac{\partial \boldsymbol{F}_t^i}{\partial k^i} \tag{2.8}$$

$$S_t^{'} = \frac{\partial S_t}{\partial F_t} \approx \frac{\alpha}{2(1+(\frac{\pi}{2}\alpha(F_t - V_{th}))^2)} = \frac{\alpha}{2(1+(\frac{\pi}{2}\alpha(\mathrm{ReLU}(kH_t) - V_{th}))^2)}$$
$$= \begin{cases} \dfrac{\alpha}{2\left(1+\left(\frac{\pi}{2}\alpha V_{th}\right)^2\right)}, & H_t < 0 \\ \dfrac{\alpha}{2\left(1+\left(\frac{\pi}{2}\alpha k\left(H_t - \frac{V_{th}}{k}\right)\right)^2\right)}, & H_t \geq 0 \end{cases} \tag{2.9}$$

In this paper, we use the derivative of arctangent $g^{'}(x) = \frac{\alpha}{2\left(1+\left(\frac{\pi}{2}\alpha x\right)^2\right)}$ in place of the derivative of the step function in Equation (2.9). α is a constant which equals 2. In Equation (2.9), the value of the surrogate gradient $S_t^{'}$ depends on the size of the parameter k. When k is large, the steepness of the surrogate gradient curve is steep; conversely, it becomes flat, as shown in

Figure 2.2(b). SNNs can adjust the gradient information by changing k during training, which is more reasonable than setting it artificially. In addition, $\frac{V_{th}}{k}$ in Equation (2.9) can be regarded as a new threshold that also depends on k. When k is large, the threshold is high; conversely, it becomes low. We use k as a shared parameter with the neurons in the same layer in SNNs. This feature not only saves memory but also is biologically plausible as the neighboring neurons tend to have similar properties [19]. Notably, the parameter k should be larger than 0, and cannot be too large as well, which leads to a very steep gradient curve. In practice, we find that the value of k rarely becomes too small or too large as a shared parameter decided by all neurons in one layer, but just in case, we still give a boundary of it from 0.5 to 5. For the initialization, we set the values of k in all layers to 1.

From one perspective, the adaptive surrogate function based on parameter k makes models more flexible during training. By optimizing the value of k, it is possible to find the best slope and width of the surrogate function, which can speed up the convergence of the model and improve the ability to fit the training data. Since each layer has a separate k, which makes the surrogate function's slopes different for each layer. From another perspective, the parameter k also scales the accumulated potential H_t at each time step. With the increase or decrease of k, the potential will be amplified or reduced. It makes the charging process of neurons more controllable.

2.2.3 Activation function ReLU

In addition to the scaling factor k, KLIF also incorporates an activation function, ReLU. ReLU keeps all the positive potentials and resets all negative potentials to 0. As k is consistently less than 1, which may cause a negative potential to be an even larger negative

value, it would reduce firing possibility and lead to dead neurons. ReLU limits the membrane potential from being too low to fire spikes. Also, the introduction of ReLU could save memory [90] for SNN quantitative representation when running on customed neuromorphic devices because 1) ReLU resets all negative potentials to zero, and 2) some gradients in backpropagation (BP) become zero due to ReLU. Figure 2.1(b) shows the feedback loop of KLIF.

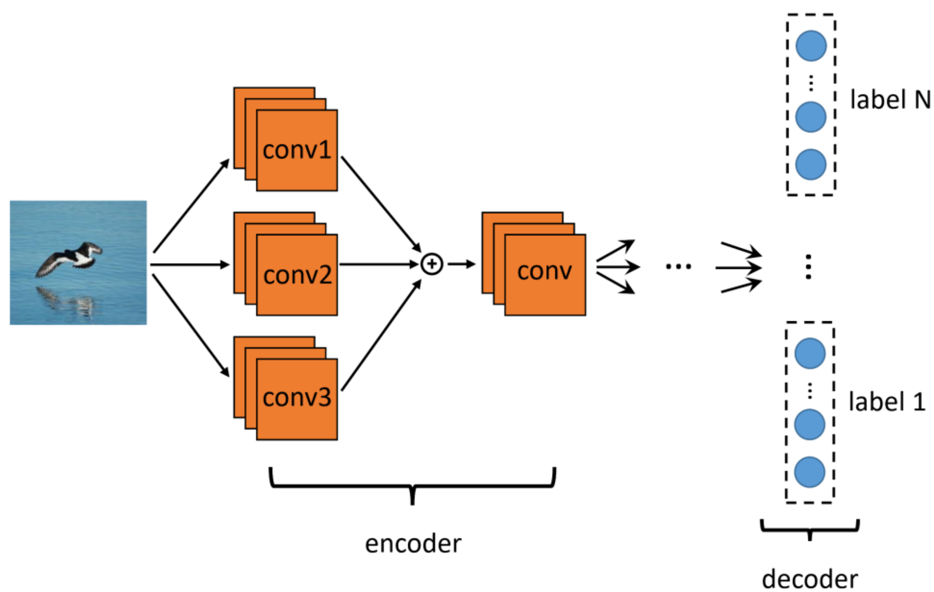


Figure 2.3 Encoder and decoder of SNNs

2.2.4 Encoding and decoding schemes

Coding layers used by SNNs are critical and decide the performance of SNNs. For the encoder, a popular method that transforms input images to spike train is rate coding. Generally, the pixel intensity of real-valued images is proportional to the firing rate in a period in rate coding. However, the conventional rate coding needs a long simulation time to present the information of images, so it is limited in training deep SNNs which have high memory requirements. An encoding layer that directly uses the first convolutional layer to encode the image information was shown to reduce the simulation time significantly and achieve a good

performance [17]. Thus, we adopt a similar method to build our encoder. The difference is that we use three parallel convolutional layers rather than one convolutional layer. We add up the values of the corresponding positions of the three convolutional layers before input to spiking neurons. We adopt a voting strategy proposed in [27] for the decoder to decode the output information. It divides neurons in the output layer into several neuron populations, and each population is assigned a label. The highest determines the output class by counting the average firing rate of every population over a given time window. Figure 2.3 shows the structure of the encoder and decoder.

Table 2.1 Network structures and training details for different datasets.

Dataset	**Simulation time**	**Epoch**	**Network structure**
MNIST and Fashion-MNIST	8	100	(128C3+128C3+128C3)(encoding)-128C3-MP2-2048FC-(100FC-AP10)(decoding)
N-MNIST	10	100	(128C3+128C3+128C3)(encoding)-128C3-MP2-2048FC-(100FC-AP10)(decoding)
CIFAR-10	10	200	(128C3+128C3+128C3)(encoding)-(256C3-256C3-256C3-MP2)*2-2048FC-(100FC-AP10)(decoding)
CIFAR10-DVS	15	100	(128C3+128C3+128C3)(encoding)-(128C3-MP2)*3-512FC-(100FC-AP10)(decoding)
DVS128-Gesture	12	200	(128C3+128C3+128C3)(encoding)-(128C3-MP2)*4-512FC- (110FC-AP10)(decoding)

Note: nC3—Convolutional layer with n output channels, kernel size = 3 and stride = 1, MP2—2D max-pooling layer with kernel size = 2 and stride = 2, AP10—1D average-pooling layer with kernel size = 10 and stride = 10, FC—FC layer. The symbol ()*n indicates the n repeated structures.

2.3 Results and discussions

We test the proposed KLIF for classification tasks on both static datasets MNIST, Fashion-MNIST, and CIFAR-10, and neuromorphic datasets N-MNIST, CIFAR10-DVS, and DVS128-Gesture. We train SNNs by the Adam[91] optimizer with the learning rate 1e-4 and the cosine annealing [92] learning rate schedule with $T_{max} = 100$.

2.3.1 Comparison of LIF and KLIF

We compare the test accuracy of SNN models on all six datasets when using LIF and KLIF, respectively (Figure 2.4). The network architectures and training details for different datasets are listed in Table 2.1. The hyperparameter selection like the number of filters and output feature maps are referenced in [19] which produces the best classification accuracy on different visual datasets. Except for the encoder, we use the same network architectures as those used in [19]. The batch normalization operation is used to change the input distribution after each convolutional layer. Before each fully connected layer, a dropout operation with the drop probability P = 0.5 is added to prevent overfitting. We keep the same hyperparameters and the network structure on both SNNs with different spiking neurons.

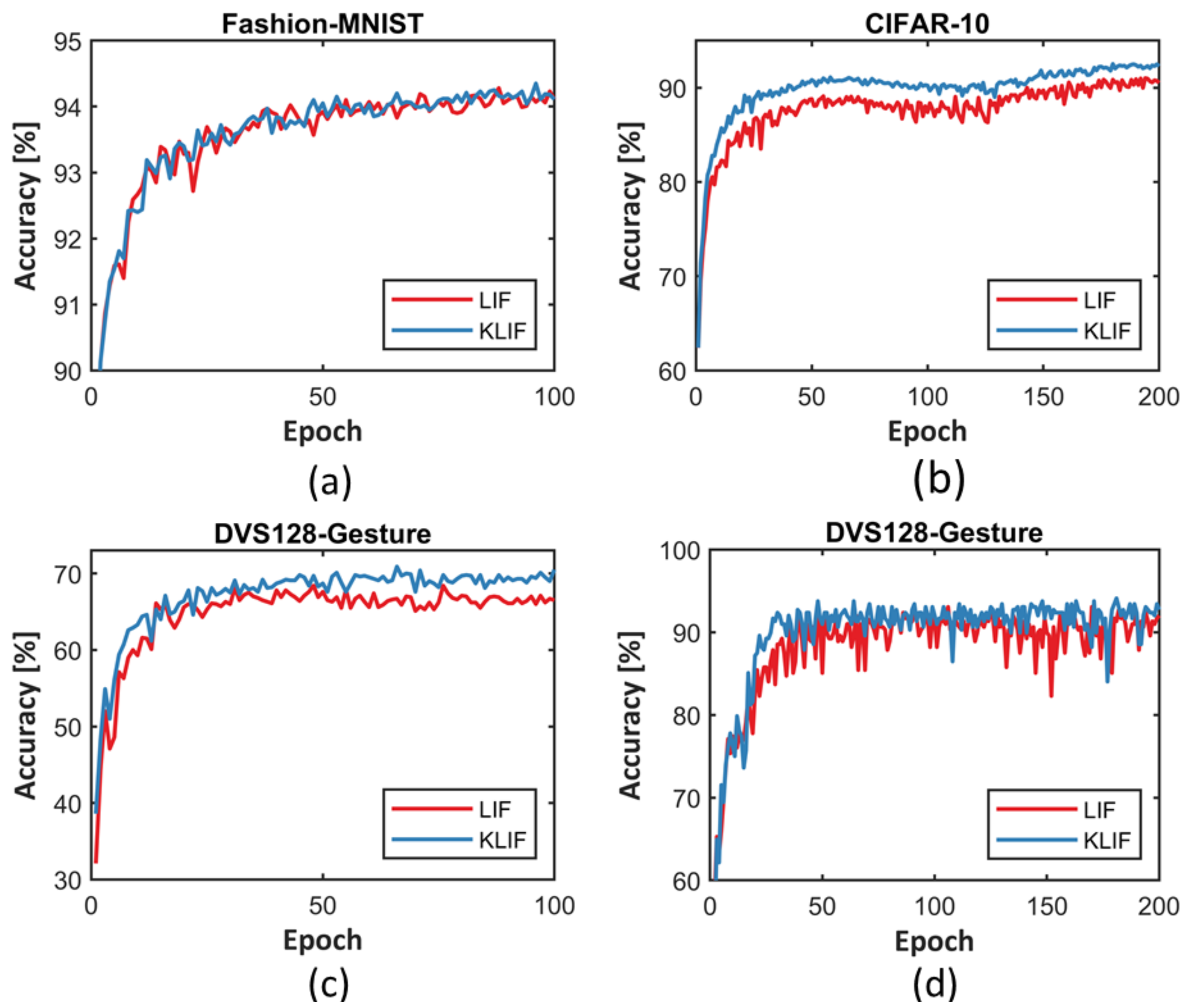


Figure 2.4 The test accuracy of KLIF v.s. LIF neurons on different datasets during training.

As shown in Figure 2.4, the test accuracy of the SNNs with KLIF neurons is always higher than that with LIF neurons, which verifies the validation of KLIF. In contrast, the accuracy gap is more significant on more complex datasets like CIFAR-10, CIFAR10-DVS, DVS128-Gesture than the simple Fashion-MNIST. Table 2.2 summarizes the results of existing state-of-the-art results. Our method achieves or approximates the best results in almost all datasets, with only no more than 10 time steps on static datasets and 15 time steps on neuromorphic datasets. Notably, accuracy listed in [19] is based on models trained for 1000 epochs, while ours are 100 or 200 epochs. With the same 100 epochs, the performance of our models is still better than that in [19] after verification.

Figure 2.5 shows the change of k in each layer during training on CIFAR-10 and DVS128-Gesture. It demonstrates that k in each layer tends to converge during training. Figure 2.6 is the distribution of firing rate comparison between the model with LIF and model with KLIF for neurons in each layer after 100 epochs training on CIFAR-10. Compared with LIF, the model with KLIF has a higher firing rate in most layers. It means that the KLIF neurons in the model are more active than LIF neurons. The result is likely to be related to the amplification of membrane potentials because most k values are larger than 1 in Figure 2.5. Besides, the presence of ReLU limits the lower bound of the membrane potential to 0, which makes it easier for neurons to accumulate to the threshold value in a short time and thus to fire spikes. Especially for LIF, in case the initial value of the membrane potential is at a small negative value, this makes a neuron hard to be triggered.

Table 2.2 Comparison between our work and the state-of-the-art methods on different datasets.

Author	Method	SNN accuracy

		MNIST	Fashion-MNIST	CIFAR-10	N-MNIST	CIFAR10-DVS	DVS128-Gesture
Hunsberger et al.[93]	ANN-SNN	98.37%	-	82.95%	-	-	-
Lucas et al.[94]	-	-	-	-	92.90%		
Rueckauer et al.[95]	ANN-SNN	99.44%	-	88.82%	-	-	-
William et am.[9]	ANN-SNN	99.53%	-	88.01%	-	-	-
Christoph et al.[96]	ANN-SNN	-	-	92.42%	-	-	-
Wu et al.[17]	Spike-based BP	-	-	90.53	99.53%	60.5%	-
Zhang et al.[97]	Spike-based BP	99.62%	90.13%	-	-	-	-
Lee et at.[16]	Spike-based BP	99.59%	-	90.95%	99.09%	-	-
Shrestha et al.[98]	Spike-based BP	99.36%	-	-	99.2%	-	93.64%
Kaiser et al.[99]	Spike-based BP	-	-	-	96%	-	95.54%
Cheng et al.[100]	Spike-based BP	99.5%	92.07%	-	99.45%	-	-
He et al.[101]	Spike-based BP	-	-	-	98.28%	-	93.40%
Xing et al[102]	Spike-based BP	-	-	-	-	-	92.01%
Wu et al.[86]	Spike-based BP	99.42%	-	-	98.78%	50.7%	-
Fang et al.[19]	Spike-based BP	99.72%	94.38%	93.5%	99.61%	74.8%	97.57%
Ours (with LIF)	Spike-based BP	99.61%	94.28%	91.02%	99.48%	68.4%	93.06%
Ours (with KLIF)	Spike-based BP	**99.61%**	**94.35%**	**92.52%**	**99.57%**	**70.9%**	**94.1%**

2.3.2 Ablation study

In section 2, we introduce the scaling factor k and ReLU function. Here we analyze their influence on models' accuracy, respectively. We selected two more commonly used network architectures for our ablation study: SNN version of VGG-16 and ResNet-18, which can demonstrate that the improved performance comes from KLIF's k and ReLU rather than a certain network architecture. We trained both networks with 100 training epochs and 6 time steps under four conditions: the accuracy of LIF, only with k, only with ReLU, and KLIF. In VGG-16, the accuracy is 78.45%, 82.96%, 81.04%, and 85.53%, respectively. In ResNet-18, a similar conclusion can be summarized. The result clearly shows the big accuracy gap

between LIF and KLIF and also indicates the benefits of k and ReLU, respectively. In contrast, using k alone achieves better results than using ReLU alone in both networks.

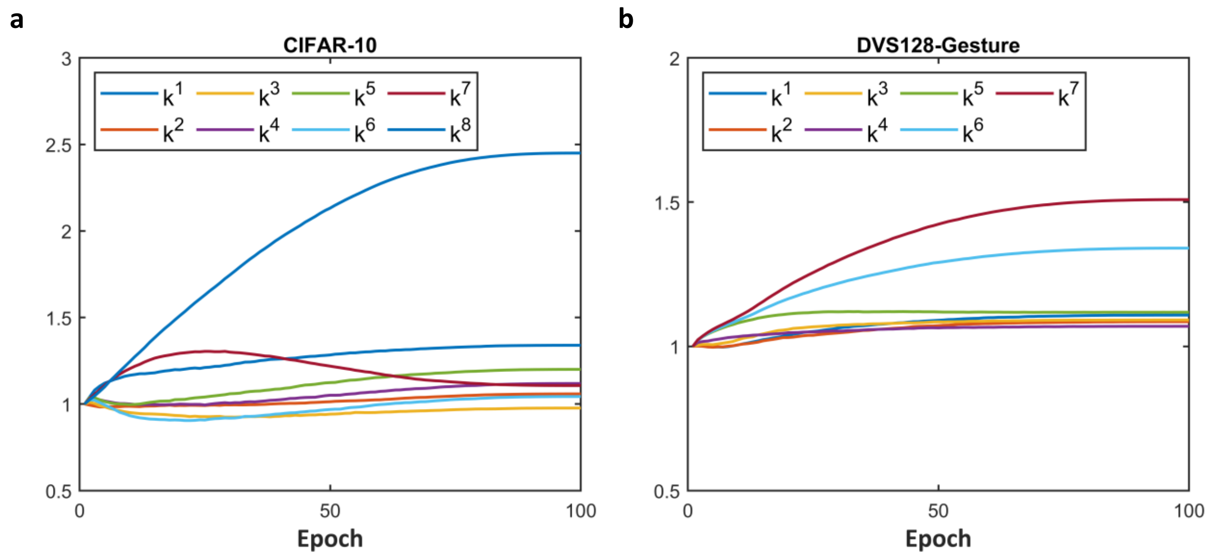


Figure 2.5 The change of scaling factor k^i in the i-th layer during training on **a.** CIFAR-10 and **b.** DVS128-Gesture.

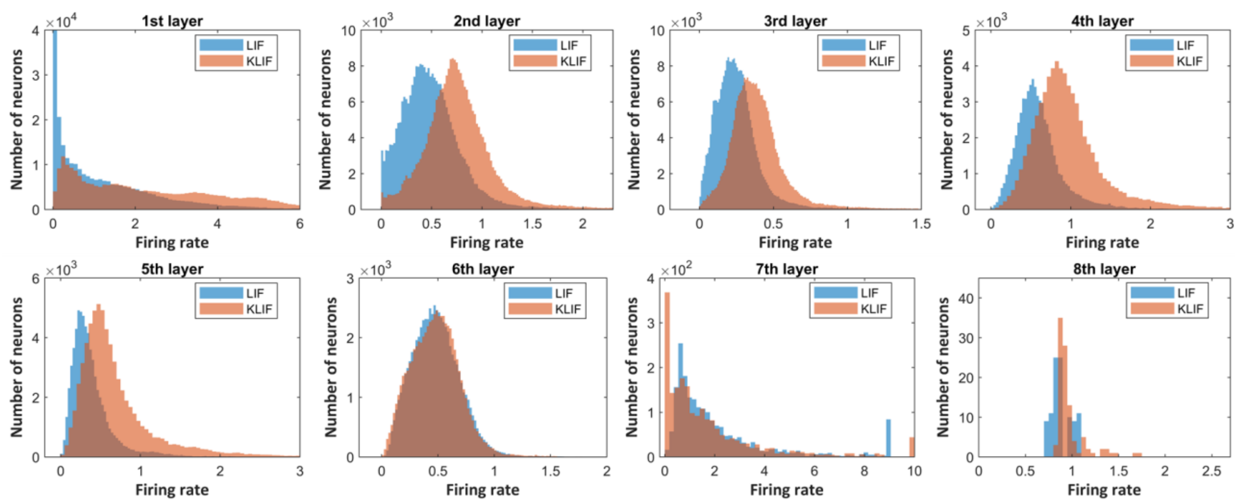


Figure 2.6 The distribution of firing rate for neurons in each layer during training on CIFAR-10.

Finally, the impact of the coding layers on the models' accuracy is tested. We incorporate our coding layers in both models. The test accuracy of the SNNs with our coding layers is always

higher than that without our coding layers, as shown in Table 2.3, showing the validity of the coding layer.

2.3.3 Biological plausibility of KLIF

Biologically, neurons regulate the ion concentration difference inside and outside the membrane through the opening and closing of ion channels, thereby regulating the magnitude and range of the membrane potential. LIF neurons cannot regulate input currents and their internal potentials during training. In contrast, the parameter k and ReLU regulate the magnitude and range of the membrane potential, respectively, which is more biologically plausible.

In Equation (2.5), F_t is a scaled and rectified version of H_t at time t. So when computing the new V_{t+1}, that will be injected in Equation (2.2) at the next time step. The scaling could also be canceled to maintain the original potential accumulation. Thus, Equation (2.7) could be changed by dividing by k on the right-hand side:

$$V_t = \frac{F_t}{k}(1 - S_t) + V_{\text{reset}} S_t = \frac{F_t}{k}(1 - S_t) \tag{2.10}$$

Table 2.3 Ablation Study of KLIF.on CIFAR-10

Neuron	VGG-16	ResNet-18
LIF	78.45%	85.24%
KLIF	**85.53%**	**89.12%**
Only with k	82.96%	87.2%
Only with ReLU	81.04%	86.66%
KLIF(without coding layers)	85.53%	89.12%
KLIF(with coding layers)	**86.64%**	**89.88%**

When we use this form as the expression of KLIF*, the accuracy (see Table 2.4) on aforementioned datasets does not change a lot compared with using KLIF.

Similarly, the ReLU function limits membrane potentials above the resting potential, which is not biologically plausible, as biological neurons can go way below the resting potential. Therefore, we replace the ReLU function in KLIF with the CELU and leaky ReLU. These functions keep the same as the ReLU in the positive part and sets the negative membrane potential as a small negative value, which is more biological plusible. The result in Table 2.5 shows that the test accuracy of SNN with KLIF(CELU) and KLIF(leaky ReLU) are still better than that with LIF. While the accuracy using KLIF(CELU) is slightly lower than that using KLIF(ReLU), the accuracy using KLIF(leaky ReLU) is not worse than that using KLIF(ReLU). The results also demonstrate the robustness of KLIF to different activation functions. In a sense, KLIF is more biologically plausible than LIF, because in LIF, the potential can be infinitely negative [46], which is inconsistent with the fact that biological

Table 2.4 Accuracy of using KLIF/KLIF*

Neuron	SNN accuracy					
	MNIST	**Fashion-MNIST**	**CIFAR-10**	**N-MNIST**	**CIFAR 10-DVS**	**DVS128-Gesture**
KLIF	99.61%	94.35%	92.52%	99.57%	70.9%	94.1%
KLIF*	99.6%	94.31%	91.93%	99.27%	70.6%	94.1%

Table 2.5 Accuracy of using KLIF with different activation functions on CIFAR-10

Neuron	VGG-16	ResNet-18
LIF	78.45%	85.24%
KLIF(ReLU)	85.53%	89.12%
KLIF(CELU)	84.55%	88.7%
KLIF(leaky ReLU)	85.9%	89.11%

neurons follow, while KLIF is more biologically meaningful by limiting the bounds of the negative values to fluctuate within a certain range through the activation function.

2.4 Conclusion

For a long time, there has been a relatively big performance gap between ANNs and SNNs. Kinds of methods like ANN-to-SNN conversion and direct training with spike-based BP attempt to reduce the gap. Overall, the spike-based BP is not as good as the conversion method regarding models' accuracy. However, the conversion from ANNs is based on the rate coding and usually needs a long inference time to approximate the original accuracy of ANNs, which is not efficient. More research is currently focused on how to train high-precision SNNs directly like ANNs.

In this work, we proposed the k-based spiking neural unit KLIF. It incorporates the learnable scaling factor k and the activation function ReLU. Our experiments show that the SNN with KLIF neurons outperforms that with LIF neurons in various visual datasets. We also verify that the scaling factor and activation function can independently contribute to improving accuracy of models. The SNN updates its learnable surrogate gradients by the scaling factor over the training. The ReLU contributes to the selective delivery of positive membrane potentials. Furthermore, our coding layers with three summed convolutional layers for SNN only needs several time steps to run, which speeds up the convergence of models and improve accuracy of models.

3 Adversarial Defense via Neural Oscillation inspired Gradient Masking

Abstract

Spiking neural networks (SNNs) attract great attention due to their low power consumption, low latency, and biological plausibility. As they are widely deployed in neuromorphic devices for low-power brain-inspired computing, security issues become increasingly important. However, compared to deep neural networks (DNNs), SNNs currently lack specifically designed defense methods against adversarial attacks. Inspired by neural membrane potential oscillation, we propose a novel neural model that incorporates the bio-inspired oscillation mechanism to enhance the security of SNNs. Our experiments show that SNNs with neural oscillation neurons have better resistance to adversarial attacks than ordinary SNNs with LIF neurons on kinds of architectures and datasets. Furthermore, we propose a defense method that changes model's gradients by replacing the form of oscillation, which hides the original training gradients and confuses the attacker into using gradients of 'fake' neurons to generate invalid adversarial samples. Our experiments suggest that the proposed defense method can effectively resist both single-step and iterative attacks with comparable defense effectiveness and much less computational costs than adversarial training methods on DNNs. To the best of our knowledge, this is the first work that establishes adversarial defense through masking surrogate gradients on SNNs.

3.1 Introduction

Spiking neural networks (SNNs) recently attracted more and more attention due to their biological plausibility. In addition to neurons and synapses, SNNs incorporate the concept of time into models. Neurons in SNN receive spike trains as inputs, and these spike trains will increase or decrease their membrane potentials. Unlike conventional artificial neural networks (ANNs), the neurons of SNNs transmit information only if their membrane potential reaches a specific firing threshold. Information is sent to the next-layer neurons in the form of spike trains. These characteristics may underline the information transmission and processing in the brain. It is therefore regarded as the next-generation neural network [103]. Like the brain working fast and efficiently, SNN is also proved to have much better power efficiency [104] and shorter latency [105] compared with ANNs. Besides, researchers also noticed their promising capability in processing dynamic and noisy information [106-108]. SNNs have been applied in various tasks such as spike pattern recognition [109], optical flow estimation [110], and sparse representation [111]. Since SNNs are being widely deployed in neuromorphic devices such as IBM TrueNorth [112] and Intel Loihi [112], the security aspect of SNNs becomes vital.

In ANNs, models are vulnerable to adversarial attacks that deceive the model into producing the wrong outputs by adding imperceptible perturbations into the clean input. This results in ANNs having catastrophic consequences in certain tasks, such as medical diagnosis and self-driving cars. These attacks are most based on gradients to generate perturbations, such as Fast Gradient Sign Method (FGSM) [67], Basic Iterative Method (BIM) [113], and Projected Gradient Descent (PGD) [114]. Therefore, it is important to improve the robustness of the model and resist the aforementioned adversarial attacks. Several adversarial defense methods

were proposed, such as ensemble training [115], denoising [116], and adversarial training [114].

Despite its popularity in ANNs, adversarial attacks rarely receive any attention in the SNN domain. One reason may be the non-differentiability of spiking events, making supervised learning of SNNs difficult. Some relevant studies of adversarial attacks on SNNs concentrate on gradient-free attack methods (e.g., trial-and-error input perturbation [117, 118]) or spatial gradient-based ANN-to-SNN conversion methods [119]. The computational complexity of the former methods is relatively high due to the absence of the gradient's guidance. The latter lacks temporal components, which leads to inefficient attacks [106]. Recently, a supervised learning algorithm using a surrogate function to approximate the derivative of spike activity [17, 86, 88] exhibited success in training high-performance SNNs and raised the opportunity to realize spatiotemporal adversarial attacks on SNNs based on gradients [120].

Adversarial defense against adversarial attacks is still in its initial stage for SNNs. There are few literatures devoted to adversarial defense methods for SNNs. In [121], the authors demonstrate that the simulation time and threshold of SNNs impact the robustness to imperceptible perturbations. However, they do not propose a defense method to resist the interference of adversarial samples effectively. In this work, we propose a specific adversarial defense method for SNN based on a novel bio-inspired approach, where neural oscillation is harnessed for the first time to enhance performances of SNNs under adversarial attacks significantly. We first present a neural oscillation neuron model to train models. The gradients of models will be masked by an alternative neural oscillation after training, thus creating interference in the gradient-based generation of the adversarial samples and effectively

enhancing the robustness of the SNNs. We have verified the effectiveness of our defense method on CIFAR-10 and CIFAR-100 datasets [122].

In summary, our main contributions are:

1. We propose a novel neural oscillation neuron that is bio-plausible and robust. It blurs the gradients of the SNNs model and interferes with the effect of perturbations on SNNs.
2. We derive an alternative neural oscillation neuron through the neural oscillation neuron. The neuron, being very 'weak', is able to attenuate the attack capability of adversarial samples, thus indirectly enhancing the robustness of the network.
3. Based on two types of neurons, we propose a defense strategy that uses the ‘fake’ neuron to confuse the attacker and thus achieve adversarial defense. The developed defense method can effectively resist kinds of adversarial attacks, such as FGSM and PGD.

The rest of this chapter is organized as follows. Section 3.2 provides some preliminaries of SNNs and adversarial attacks. The experimental setup and our neural oscillation models are discussed in Section 3.3. Section 3.4 validates the validation of our defense methodology. Section 3.5 concludes this article.

3.2 Preliminaries

3.2.1 SNNs and biological neural oscillation

SNNs adopt spike trains as information carriers between neurons. Every spiking neuron in a SNN receives and emits spikes. The LIF neuron model is a popular bio-inspired simplified model for describing the dynamics of spiking neurons. The dynamics of the LIF model are defined [19] by

$$H(t) = \lambda * V(t-1) + \sum_i w_i x_i(t)$$

(3.1)

$$S(t) = \begin{cases} 1, & H(t) > V_{th} \\ 0, & H(t) \leq V_{th} \end{cases}$$

(3.2)

$$V(t) = H(t)\big(1 - S(t)\big) + V_{\text{reset}} * S(t)$$

(3.3)

where $H(t)$ and $V(t)$ represent the membrane potentials before and after triggering a spike at time t, respectively. V_{th} denotes the firing threshold, which is 1 in this paper. V_{reset} is the resting potential, which is 0. S_t denotes the output of neurons at time t, $w_i x_i(t)$ is the i-th weighted pre-synaptic input at time t, and λ is the decaying time constant, which is 0.5.

According to Equations (3.1) - (3.3), when a neuron receives spikes from the previous-layer neurons, its membrane potential will increase. Once the potential value surpasses the neuron's firing threshold, the neuron will fire one spike and promptly be reset to the initial potential V_{reset}.

The biological nervous system generates rhythmic patterns of activity called neural oscillation [123]. Neural oscillations are thought to be associated with many cognitive functions such as information transfer, perception, motor control, and memory. Such oscillation is mainly triggered by the interaction of individual neurons. In individual neurons, neural oscillation can manifest as the oscillation of membrane potentials or as rhythmic action potentials. This kind of spontaneous activity plays an important role during brain development, including synaptogenesis and network formation. Even though neural oscillations are ubiquitous in

biological neurons, the current common spiking neuron models for deep learning, such as IF and LIF models, do not include this oscillatory mechanism, and there is no literature that develops adversarial defense of deep learning models using neural oscillation.

3.2.2 Adversarial attacks

Adversarial attacks [124] introduce imperceptible perturbation into the input data to mislead the model's classification result. Adversarial attacks can be classified as targeted and non-target attacks according to adversarial goals. A targeted attack is when the attacker attempts to misdirect the model to a class that is different from the true class, while a non-target attack means that the attacker attempts to mislead the model by predicting any of the incorrect classes [125].

In gradient-based adversarial attacks, for a clean image x belonging to class k and a trained SNN model M, the adversarial image $x_{adv}$of x needs to satisfy the following two criteria:

1). The difference between x_{adv} and x is imperceptible, i.e., $\left||x_{adv} - x|\right|_p \leq \epsilon$

2). The model misclassifies x_{adv}, i.e., $M(x_{adv}) \neq k$

where the distance metric $||.||_p$ denotes the p-norm quantifying the similarity, and ϵ reflects the maximum allowable perturbation on the image.

There are various kinds of adversarial attack algorithms that generate adversarial samples to deceive the model. In this work, we adopt four typical adversarial attacks to evaluate our defense model.

Fast Gradient Sign Method (FGSM) [67] is the most basic approach for generating adversarial samples, which aims at finding a perturbation that maximizes its cost function for

the perturbed input [120]. This approach generates adversarial samples by perturbing once the clean image x by the amount of ϵ along the input gradient direction:

$$\boldsymbol{x_{adv}} = \boldsymbol{x} + \epsilon \cdot \text{sign}\big(\nabla_x \mathcal{L}(\boldsymbol{x}, \boldsymbol{y})\big) \tag{3.4}$$

Here, $\mathcal{L}$ represents the cost function of the model, and $\nabla_x(*)$ is the model's gradient with respect to a clean sample of $\boldsymbol{x}$. $\boldsymbol{y}$ is the label corresponding to $\boldsymbol{x}$.

Basic Iterative Method (BIM) [113] is an iterative version of FGSM and generates the adversarial samples as:

$$\boldsymbol{x_m} = \text{clip}_\epsilon\left(\boldsymbol{x_{m-1}} + \frac{\epsilon}{i} \cdot \text{sign}\left(\nabla_{x_{m-1}}\big(\mathcal{L}(\boldsymbol{x_{m-1}}, \boldsymbol{y})\big)\right)\right) \tag{3.5}$$

where $\boldsymbol{x_0}$ is the clean image, $\boldsymbol{x_m}$ is an adversarial sample in the m-th iteration, and i is the iteration number. $\text{clip}_\epsilon(*)$ represents element-wise clipping of the argument to the range $[\boldsymbol{x} - \epsilon, \boldsymbol{x} + \epsilon]$.

Momentum Iterative Method (MIM) [126] is similar to BIM but is extended to promote the stability of gradient direction through the addition of a momentum term:

$$g_m = \mu \cdot g_{m-1} + \frac{\nabla_{x_{m-1}} \mathcal{L}(\boldsymbol{x}_{m-1}, y)}{||\nabla_{x_{m-1}}\big(\mathcal{L}(\boldsymbol{x}_{m-1}, y)\big)||_1} \tag{3.6}$$

$$\boldsymbol{x}_m = \text{clip}_\epsilon\left(\boldsymbol{x}_{m-1} + \frac{\epsilon}{i} \cdot \text{sign}(g_m)\right) \tag{3.7}$$

μ is the decaying factor.

Projected Gradient Descent (PGD) is one of the strongest iterative adversary attacks. It starts from a random position in the clean image neighborhood $\mathcal{U}(\boldsymbol{x}, \epsilon)$. Its expression is described as:

$$\boldsymbol{x}_m = clip_\epsilon\left(\boldsymbol{x}_{m-1} + \gamma \cdot sig\,n\left(\nabla_{x_{m-1}}\mathcal{L}(x_{m-1}, y)\right)\right) \tag{3.8}$$

where m is the iterative number, and γ is the step size.

3.3 Experiments

3.3.1 Datasets and Models

We conduct the experiments on SNN versions of VGG-16 and ResNet-18 for CIFAR-10 and ResNet-18 for CIFAR-100. All models were trained by surrogate gradient-based BP with maxpool layers replaced by average pooling. Bias terms are not included in SNNs. After the convolution layer, we add a batch normalization layer to change the input distribution. Before the fully connected layer, a dropout layer with the probability of P = 0.5 is used to prevent overfitting.

For both CIFAR-10 and CIFAR-100 datasets, all data are normalized to [0,1]. SNNs are trained for 100 epochs with cross-entropy loss and Adam [91] optimizer. The initial learning rate is set to 1e-4, and the cosine annealing [92] learning rate schedule with T_{max} = 100 adjusts the learning rate over training. A total of 8 timesteps are used for all SNNs. We measure the attack success rate of adversarial sample crafting on 1000 samples randomly selected from each dataset.

Due to the discontinuity of the spiking activity, when training the model, we use the derivative of the Atan function as the surrogate gradient function (see Equation (3.9), $\alpha = 3$) to approximate the derivative of spiking activities.

$$y(x) = \frac{\alpha}{2\left(1 + \left(\frac{\pi}{2}\alpha(x - V_{th})\right)^2\right)} \tag{3.9}$$

3.3.2 Neural oscillation neuron

Inspired by the biological neural oscillation, we add random oscillation noise in the LIF

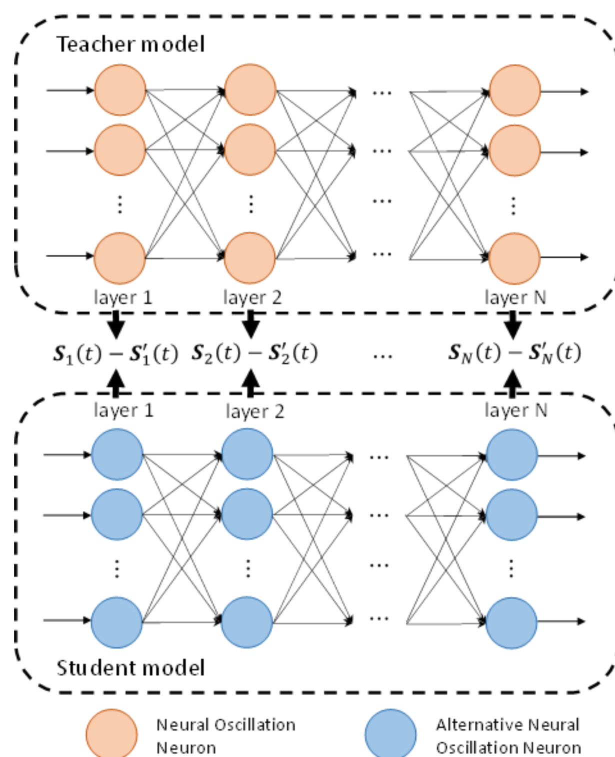


Figure 3.1 The training process of the model with alternative neural oscillation neurons at time t. The model trained first with neural oscillation neurons can be regarded as a 'teacher model'. It provides the labels for a 'student model' which replaces neural oscillation neurons with alternative neural oscillation neurons. The 'student model' keeps the same trained weights and fits spike trains $S'_j(t)$ of student model to $S_j(t)$ of teacher model by learning variables a and b in each layer.

neuron. We refer to the new neuron as the neural oscillation model. Its dynamic can be described by Equations (3.1) and (3.10) - (3.13).

$$P(t) = f(H(t) + \gamma(t)) \tag{3.10}$$

$$S(t) = \begin{cases} 1, & P(t) > V_{th} \\ 0, & P(t) \le V_{th} \end{cases} \tag{3.11}$$

$$V_t = P(t)\big(1 - S(t)\big) + V_{\text{reset}} S(t) \tag{3.12}$$

$$f(x) = \begin{cases} -0.03x, & x < 0 \\ x, & x \ge 0 \end{cases} \tag{3.13}$$

$\gamma(t)$ is an independent uniformly-distributed random noise in a range of $[a, b]$ for neurons in each layer, which is [-0.2,0.8] in this paper. $f(x)$ is a piece-wise linear function $LeakyReLU$ whose gradients are defined as -0.03 if $x \le 0$ and gradients are 1 if $x > 0$.

3.3.3 Alternative Neural oscillation neuron

We train and then save SNN with neural oscillation model, then we copy the model and replace the neural oscillation neuron with a new neural model called alternative neural oscillation. The new neural model changes the noise item $\gamma_i(t)$ in Equation (3.10) to a Sine function of the membrane potential $H(t)$, as Equation (3.14) describes. The firing and reset actions keep the same as Equations (3.11) and (3.12).

$$P(t) = f(H(t) + \sin(H(t) + c) + d) \tag{3.14}$$

Variables c and d are learnable parameters shared by all layers in the network. The mapping function $\sin(H(t)+c)+d$ is selected here to fit the noise in neural oscillation (More details of the mapping function selection can be found in the supplementary). We freeze all weights of the model with alternative neural oscillation and only keep parameters c and d learnable. The model as the student model was trained again to learn each layer's output of the saved model with neural oscillation neurons, which is regarded as the teacher's model. Here we define the loss function Equation (3.15) to minimize the difference between the spike trains $\boldsymbol{S}_j(t)$ and $\boldsymbol{S'}_j(t)$ between neural oscillation and alternative neural oscillation in j-th layer at time t.

$$L = \sum_t \sum_j \frac{1}{2}(\boldsymbol{S}_j(t) - \boldsymbol{S}'_j(t))^2 \tag{3.15}$$

In this way, the accuracy of models changes slightly (see Table 3.1). In experiments this process requires only 1-3 training epochs, besides, weight parameters are frozen and few parameters are learnable, thus adding almost no additional training time. The details of noise range $[a, b]$ and values of c and d obtained by training can be found in the supplementary. The process of generating alternative neural oscillation neurons is presented in Figure 3.1.

3.3.4 Adversarial defense strategy

Now we have two models with different neurons. The two models have approximate output and inference accuracy but with different gradients. For neural oscillation model, we calculate the gradients $\frac{\partial S(t)}{\partial H(t)}$, when $H(t)+\gamma_i(t)\geq 0$,

$$\frac{\partial S(t)}{\partial H(t)} = \frac{\partial S(t)}{\partial P(t)}\frac{\partial P(t)}{\partial H(t)} \approx \frac{\alpha}{2\left(1+\left(\frac{\pi}{2}\alpha(H(t)+\gamma_i(t)-V_{th})\right)^2\right)} \tag{3.16}$$

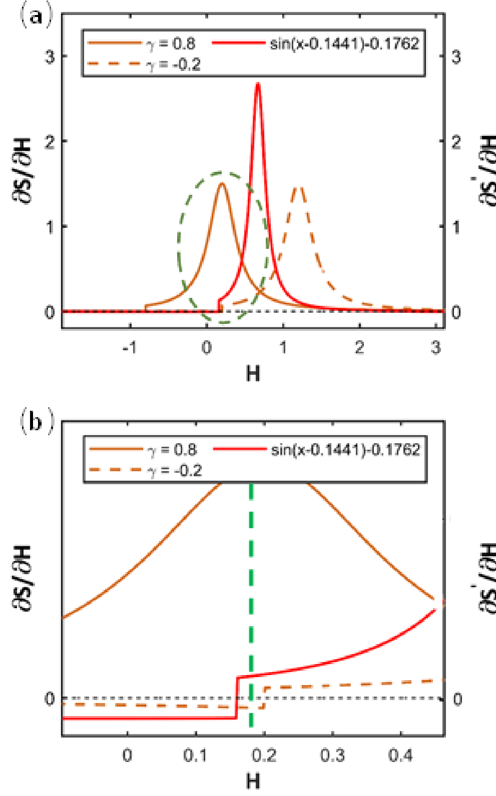


Figure 3.2 (a). The solid and dotted orange lines represent $\frac{\partial S}{\partial H}$ of neural oscillation model when γ is -0.2 and 0.8, respectively. The red line is $\frac{\partial S'}{\partial H}$ of alternative neural oscillation model. (b). Partial enlargement of graph (a) in the green dashed circle.

when $H(t) + \gamma_i(t) < 0$,

$$\frac{\partial S(t)}{\partial H(t)} = \frac{\partial S(t)}{\partial P(t)}\frac{\partial P(t)}{\partial H(t)} \approx \frac{-0.03\alpha}{2\left(1 + \left(\frac{\pi}{2}\alpha(H(t) + \gamma_i(t) - V_{th})\right)^2\right)} \tag{3.17}$$

For alternative neural oscillation model, when $H(t) + \sin(H(t) + a) + b \geq 0$, the gradients $\frac{\partial S'(t)}{\partial H(t)}$ is

$$\frac{\partial S'(t)}{\partial H(t)} = \frac{\partial S'(t)}{\partial P(t)}\frac{\partial P(t)}{\partial H(t)} \approx \frac{\alpha(1 + \cos(H(t) + a))}{2\left(1 + \left(\frac{\pi}{2}\alpha(H(t) + \sin(H(t) + a) + b - V_{th})\right)^2\right)}$$

$$(3.18)$$

when $H(t) + \sin(H(t) + a) + b < 0$,

$$\frac{\partial S'(t)}{\partial H(t)} = \frac{\partial S'(t)}{\partial P(t)} \frac{\partial P(t)}{\partial H(t)} = \frac{-0.03\alpha(1 + \cos(H(t) + a))}{2\left(1 + \left(\frac{\pi}{2}\alpha(H(t) + \sin(H(t) + a) + b - V_{th})\right)^2\right)}$$

$$(3.19)$$

We plot both models' gradient distributions of VGG16 for CIFAR-10. As we see, the gradient distribution using alternative neural oscillation is distinguished from the gradient distribution using neural oscillation. The gradient distribution of the alternative neural oscillation has the bigger amplitude and the sharper shape than that in Figure 3.2(a). On the other hand, in Figure 3.2(b), we zoom in on the part of the dashed circle in Figure 3.2(a) and observe the gradient, and it could be seen that gradients vary greatly at the same H value between neural oscillation neuron and alternative neural oscillation neuron.

Our defense strategy is to disguise and hide the real neurons in the model, which confuses the attacker to generate attack samples and attack with the gradients of the 'fake' neurons, which deviate from the real gradients and thus reduce the efficiency of the attack. We validate the effectiveness of this defense method based on the alternative neural oscillation in Section 3.4.

3.4 Results

We train SNN with the LIF model as the benchmark model to facilitate the comparison of the validation of our approach. The maximum perturbation size ε in all attacks is 8/255. For iterative attacks, for example, PGD-i, i indicates the number of iterative steps. All attacks are non-target attacks.

3.4.1 Robustness analysis of neural oscillation model

We first test the robustness of the neural oscillation model against adversarial attacks. As shown in scenario 1 of Table 3.2, when attackers are fully aware of the structure, parameters, and form of the neural oscillation model, they use adversarial samples to attack networks. The results are presented in Table 3.3. Neural oscillation neuron always performs better than LIF neuron in terms of robustness in all three models/datasets under four attacks. The reason is that we introduce randomization. When noise is added to the neurons, this interferes with the validity of the perturbations superimposed in the input images. As illustrated in Figure 3.2(a), when noise γ_i is different values, the gradient curve changes accordingly. This causes gradient value blurring and reduces the effectiveness of the generation of adversarial samples. Besides, $LeakyReLU$ leads gradients in specific sections to be small negative values, which makes it possible to cause the gradient's direction to blur. For example, in Figure 3.2(b), when $\mathrm{H} = 0.18$ (green dashed line), the gradient of neuron with $\gamma_i = 0.8$ is positive, while the gradient of neuron with $\gamma_i = -0.2$ is negative. Thus, this randomness includes not only randomization in the value of the gradient but also the direction of the gradient.

3.4.2 Robustness analysis of alternative neural oscillation model

We then investigate the performance of alternative neural oscillation model with respect to robustness. As indicated in scenario 2 of Table 3.2, when attackers are fully aware of the structure, parameters, and form of the alternative neural oscillation model, they use adversarial samples to attack networks. The experimental results in Table 3.4 suggest that the alternative neural oscillation neurons have better robustness than LIF neurons under different adversarial attacks. And they are also more robust compared to neural oscillation neurons. The reason is that the alternative neural oscillation neuron replaces the random noise as a function of $H(t)$, which leads gradients $\frac{\partial s'(t)}{\partial H(t)}$ to be too steep (see Figure 3.2(a)). The steep gradients make the network back propagation optimization parameters unstable (gradient vanishing and

exploding). According to the definitions of attack methods in section 3.2, the generation of adversarial samples depends entirely on the exact gradient information so the unstable gradient makes the generated adversarial samples less aggressive and reduce the effectiveness of adversarial attacks. We have placed extra experimental results and argued this conclusion in the supplementary material.

Table 3.1 Top-1 Accuracy (%) on clean images using two kinds of oscillation neurons

Model/Dataset	Neural oscillation	Alternative neural oscillation	Accuracy loss
VGG-16/CIFAR-10	88.59	87.38	1.21
ResNet-18/CIFAR-10	92.59	92.35	0.24
ResNet-18/CIFAR-100	67.58	67.05	0.53

Table 3.2 Neuron model summary under different attack scenarios

<table>
<tr><th>Scenario</th><th>Attackers know the real neuron model</th><th>Neuron model chosen by attackers to generate adversarial samples</th><th>The real neuron model used for inference</th></tr>
<tr><td>Scenario 1</td><td rowspan="2">Yes</td><td>Neural oscillation</td><td>Neural oscillation</td></tr>
<tr><td>Scenario 2</td><td>Alternative neural oscillation</td><td>Alternative neural oscillation</td></tr>
<tr><td>Scenario 3</td><td rowspan="3">No</td><td>LIF</td><td>Neural oscillation</td></tr>
<tr><td>Scenario 4</td><td>Alternative neural oscillation</td><td>Neural oscillation</td></tr>
<tr><td>Scenario 5</td><td>Neural oscillation</td><td>Alternative neural oscillation</td></tr>
</table>

Table 3.3 Top-1 classification accuracy (%) under the scenario 1 attack

Models/Datasets	VGG-16/CIFAR-10		ResNet-18/CIFAR-10		ResNet-18/CIFAR-100	
	benchmark	ours	benchmark	ours	benchmark	ours
Clean	88.6	88.59	92.2	92.59	65.7	67.58
FGSM	14.2	30.9	35.5	44.9	12.6	17.2
PGD-5	3	14.7	5.8	21.7	1.3	4.3
BIM-5	1.9	14.3	5.9	21.4	1.4	4.5
MIM-5	2.8	14.9	8.8	22.3	1.6	3.1

Table 3.4 Top-1 classification accuracy (%) under the scenario 2 attack

Models/Datasets	VGG-16/CIFAR-10		ResNet-18/CIFAR-10		ResNet-18/CIFAR-100	
	benchmark	ours	benchmark	ours	benchmark	ours
Clean	88.6	87.38	92.2	92.35	67.04	67.05
FGSM	14.2	52.2	35.5	66.9	12.6	45.1
PGD-5	3	28.1	5.8	57.1	1.3	39.3
BIM-5	1.9	31.6	5.9	59	1.4	39.8
MIM-5	2.8	30.8	8.8	53.4	1.6	33.2

Table 3.5 Top-1 classification accuracy (%) under the scenario 3 attack

Models/Datasets	VGG-16/CIFAR-10		ResNet-18/CIFAR-10		ResNet-18/CIFAR-100	
	benchmark	ours	benchmark	ours	benchmark	ours
Clean	88.6	88.59	92.2	92.59	67.04	67.58
FGSM	14.2	40.1	35.5	71.9	12.6	47.7
PGD-5	3	40.8	5.8	79.3	1.3	56
BIM-5	1.9	36.2	5.9	71.8	1.4	56.7
MIM-5	2.8	26.7	8.8	67.6	1.6	49.4

Table 3.6 Top-1 classification accuracy (%) under the scenario 4 attack

Models/Datasets	VGG-16/CIFAR-10		ResNet-18/CIFAR-10		ResNet-18/CIFAR-100	
	benchmark	ours	benchmark	ours	benchmark	ours
Clean	88.6	88.59	92.2	92.59	67.04	67.58
FGSM	14.2	56.5	35.5	69.5	12.6	43.7
PGD-5	3	45	5.8	66.8	1.3	48.4
BIM-5	1.9	41.4	5.9	68.5	1.4	51.4
MIM-5	2.8	35.4	8.8	59	1.6	40.8

Table 3.7 Top-1 classification accuracy (%) under the scenario 5 attack

Models/Datasets	VGG-16/CIFAR-10		ResNet-18/CIFAR-10		ResNet-18/CIFAR-100	
	benchmark	ours	benchmark	ours	benchmark	ours
Clean	88.6	87.38	92.2	92.35	67.04	67.05
FGSM	14.2	28.4	35.5	42.1	12.6	13.8
PGD-5	3	12.1	5.8	19.2	1.3	4
BIM-5	1.9	12.5	5.9	19.4	1.4	2.6
MIM-5	2.8	13	8.8	20.2	1.6	2.4

Table 3.8 White-box robustness (accuracy (%)) on CIFAR-10 using the ResNet-18 (ϵ=8/255)

Defense	Clean	FGSM	PGD-20

Standard [114]	84.44	61.89	47.55
MMA [71]	84.76	62.08	48.33
Dynamics [127]	83.33	62.47	49.40
TRADES [73]	82.90	62.82	50.25
MART [74]	83.07	65.65	55.57
Ours (scenario 4)	**92.59**	**69.5**	**71.1**

3.4.3 Validation of defense

Scenarios 1 and 2 are both white-box attacks, where attackers are fully aware of all information about the network. Sometimes attackers only know part of the network, i.e. a grey-box attack. In this section, we test the validity of our defense strategy of masking real neurons with false neurons, thus tricking the attacker into generating attack samples with bias and reducing the efficiency of the attack. We consider three scenarios which are summarized in Table 3.2:

Scenario 3: Attackers are aware of the structure and parameters but do not know the form of neural model. Attackers use the LIF model to generate adversarial samples; however, neural oscillation is the real neural model for inference.

Scenarios 4 and 5: Attackers are aware of structure, parameters and know either the form of neural oscillation or the form of alternative neural oscillation. Attackers use the known 'fake' neurons to generate adversarial samples. The other neural model is as the real neuron of network for inference.

For scenario 3, Table 3.5 shows the top-1 accuracy of both benchmark and our method. The result demonstrates that our defense performance for both single-step attacks and iterative attacks on all three models is significantly better than SNNs. Since attackers do not know the specific expression of the neuron, this causes a significant decrease in the attack efficiency of the generated sample perturbations. For scenario 4, similar results to scenario 3 in Table 3.6 are observed: our method performs much better than benchmark against both single-step attack and iterative attack in all models and datasets. The results indicate that gradients of alternative

neural oscillation neurons lead to a decreased attack success rate by masking original training gradients. For scenario 5, our method is still more robust to adversarial samples than benchmark in Table 3.7. However, it is worth noting that the defense of scenario 5 is much weaker when comparing the defense ability of scenario 4. When attackers know the expression of neural oscillation neuron, even though we replace them with the alternative neurons for inference, attackers can still generate effective adversarial samples to attack our networks. In other words, when attackers use alternative neural oscillation neuron to generate adversarial samples, the neuron, being very 'weak', is able to attenuate the attack capability of adversarial samples, thus indirectly enhancing the robustness of networks.

In fact, if we directly discard the neural oscillation neuron after training the model and replace it with the alternative neural oscillation neuron, then deploy the model in hardware devices, it is easy for attackers to be fooled by the ‘fake’ neuron.

Table 3.8 compares alternated neural oscillation with some advanced adversarial training defense methods in ANN. The adversarial training requires a large number of samples to retrain the model, and it is impractical to introduce all unknown attack samples into adversarial training, which would consume much time and computational resources, leading to the limitation of adversarial training. Our method only requires additional learning of a new oscillatory form through introducing only two parameters, which defends against most gradient-based adversarial attacks and is more efficient.

As neural oscillations are essential to many neural activities in the biological nervous system, SNN integrated with oscillation mechanism is more bio-plausible (In the supplementary we shows the spontaneous spike firing of our neural oscillation model, which is similar to the

biological neural oscillation). Various mechanisms of the biological neural system provide a basis for optimizing the SNN, while these mechanisms integrated into the SNN also help us to better understand the biological neural system.

3.5 Discussion and conclusion

In this chapter, we integrate brain-inspired neural oscillation into the SNN neural model and propose the neural oscillation neural model and alternative neural oscillation neural model. We verified that both neural models have better robustness than the LIF neuron. And we also use alternative neural oscillation neuron as the 'fake' neuron to defend against various gradient-based attacks. The experiments illustrate that our defense method can effectively resist both single-step attack and iterative attack.

Our method belongs to the class of methods that introduce randomization to enhance network robustness, but it is very different from the randomization currently used in ANNs. While previous literatures, such as [70], usually use random perturbations to disturb the generated samples, our method only introduces randomization over training and it is replaced by fitting a specific function during inference, which causes instability of the gradient. Therefore, our trained model has no randomization after training process, and the advantage of this approach is that when the attacker is fully aware of the neuronal model, the previous defense method only needs to remove the randomization to achieve an effective attack model, while our method cannot effectively attack the model after removing the fitting function. The attacker must work harder to find the original noise distribution in order to attack the model effectively. Thus, our method is more deceptive, which will make it difficult for the attacker to detect anomalies in the network.

There are still some limitations to our method. Since most of the current attacks are gradient-based attacks, our defense method was originally developed based on gradient-based attacks, so we did not test its effectiveness on other attack types. This part could be further explored in the future. Certainly, the neural oscillation model only partially mimics the form of biological neural oscillation; thus, further research might be conducted to integrate more complex neural oscillation forms in SNNs.

3.6 Supplementary

3.6.1 Parameter values for reproducibility

Table 3.9 shows the noise range and parameters c and d. In our work, we picked [a,b] in the range [-0.2,0.8]. The noise is generally selected not to exceed the threshold V_{th} (otherwise it may lead to a reduction of accuracy) and is mainly concentrated between V_{th} and the V_{reset}.There is no mandatory range size for the selection of these hyperparameters, and we actually tried different ranges and were able to obtain similar defensive effects. For noise type, in the main paper we use the random uniform noise, we have also tried the Gaussian noise, and it can also be fitted by different equations to generate the alternative neural oscillation neuron.

Table 3.9 Noise range [a,b] and values of c and d of alternative neural oscillation model

Model/Dataset	$[a, b]$	c	d
VGG-16/CIFAR-10	[-0.2,0.8]	-0.1441	-0.1762
ResNet-18/CIFAR-10		-0.1019	-0.2221
ResNet-18/CIFAR-100		-0.1564	-0.1687

3.6.2 Neuron performance testing

Figure 3.3 depicts the accuracy of the SNNs on the corresponding architecture/dataset when using either neural oscillation neuron or alternative neural oscillation neuron. In Figure 3.3(a), the VGG-16 network composed of alternative neural oscillation neurons cannot even be optimized. In Figure 3.3(b), the alternative neural oscillation neuron significantly decreases the network optimization's speed and accruracy. Thus, the results indicate that alternative neural oscillatory neurons have worse performance than neural oscillatory neurons, which tend to lead to instability of the gradient (e.g. gradient vanishing or explosion), and hence make the network less capable of optimization. When an attacker generates the adversarial samples with such neurons, the adversarial samples also become less powerful.

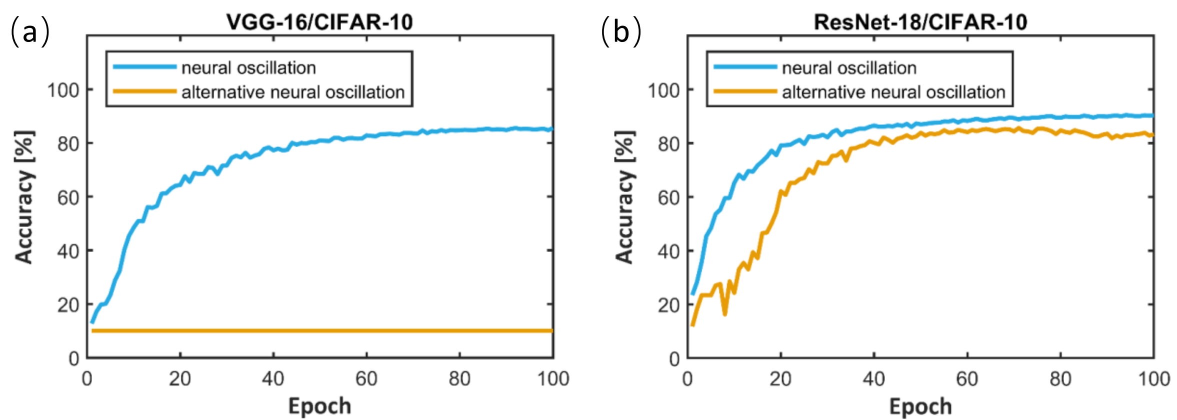


Figure 3.3 Accuracy on (a)VGG-16/CIFAR-10 (b)ResNet-18/CIFAR-10 when using neural oscillation neuron (blue curve) and alternative neural oscillation neuron (orange curve)

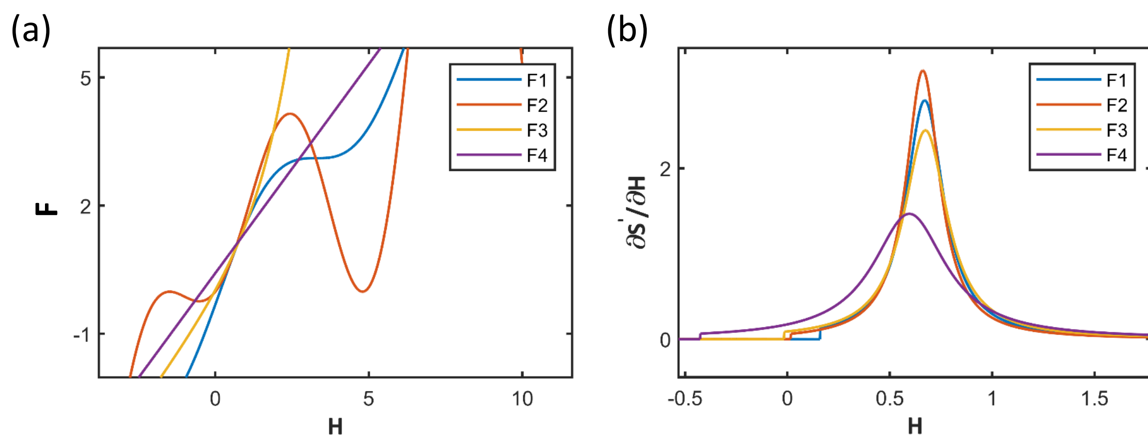


Figure 3.4 (a) Curve of function F to fit the noise item on VGG-16/CIFAR-10. (b) Gradient curve $\frac{\partial S'(t)}{\partial H(t)}$ when using different F.

Table 3.10 Top-1 classification accuracy (%) under the scenario 4 attack

Models/Datasets	VGG-16/CIFAR-10				
	benchmark	*F1*	*F2*	*F3*	*F4*
Clean	88.6	87.38	86.12	87.7	86.64
FGSM	14.2	56.5	68.9	50.7	36.8
PGD-5	3	45	68.8	34.6	22.7
BIM-5	1.9	41.4	69.3	32.9	15
MIM-5	2.8	35.4	60.4	31.4	20.6

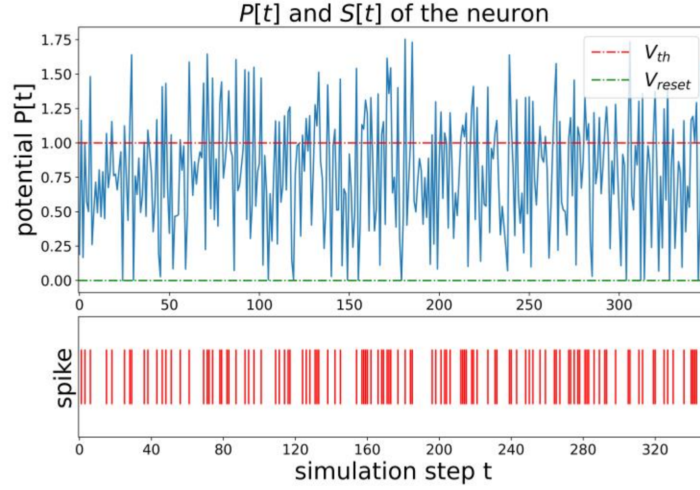


Figure 3.5 Spontaneous spike firing of neural oscillation neuron.

3.6.3 Function selection of alternative neural oscillation

In the main paper, we use the $\sin(H(t) + c) + d$ to fit the random uniform noise item $\gamma(t)$. In practice, the function can be of many different forms, such as Equations (3.20)-(3.23). These equations all fit the random noise term $\gamma(t)$ well after training parameters c and d.

$$F1 = sin(H(t) + c_1) + d_1 \tag{3.20}$$

$$F2 = x * sin(H(t) + c_2) + d_2 \tag{3.21}$$

$$F3 = e^{(x+c_3)} + d_3 \tag{3.22}$$

$$F4 = \frac{1}{1 + e^{-c_4 x}} + d_4$$

(3.23)

We draw the *F* curves in Figure 3.4(a) and the corresponding gradient curves Figure 3.4(b) when different *F* is chosen, respectively. Table 3.10 compares the effectiveness of the defense when using different *F*. As we see, *F2* provides the best defense against all kinds of attacks, while *F4* is the worst. And the gradient curve of *F2* shown in Figure 3.4(b) is the steepest, when the gradient curve of *F4* is the flattest. These results argue our view that the steep gradient causes instability in the network, which weakens the effectiveness of generated adversarial samples.

3.6.4 Firing property of neural oscillation neuron

Neural oscillation arises from the spontaneous spike firing behavior of biological neurons. This spontaneous behavior is not influenced by external stimuli. In our proposed neural oscillation neural model, the inclusion of random noise allows the neuron to generate spontaneous spike firing in the absence of input, as shown in Figure 3.5. This property makes our model more bio-plausible.

4 A noise based novel strategy for faster SNN training

Abstract

Spiking neural networks (SNNs) are receiving increasing attention due to their low power consumption and strong bio-plausibility. Optimization of SNNs is a challenging task. Two main methods, artificial neural network (ANN)-to-SNN conversion and spike-based backpropagation (BP), both have their advantages and limitations. For ANN-to-SNN conversion, it requires a long inference time to approximate the accuracy of ANN, thus diminishing the benefits of SNN. With spike-based BP, training high-precision SNNs typically consumes dozens of times more computational resources and time than their ANN counterparts. In this chapter, we propose a novel SNN training approach that combines the benefits of the two methods. We first train a single-step SNN by approximating the neural potential distribution with random noise, then convert the single-step SNN to a multi-step SNN losslessly. The introduction of Gaussian distributed noise leads to a significant gain in accuracy after conversion. The results show that our method considerably reduces the training and inference times of SNNs while maintaining their high accuracy. Compared to the previous two methods, ours can reduce training time by 65%-75% and achieves more than 100 times faster inference speed. We also argue that the neuron model augmented with noise makes it more bio-plausible.

4.1 Introduction

Spiking neural networks (SNNs) recently attracted increasing attention due to their biological plausibility. The SNN incorporates the concept of time into the model, and neurons in the SNN receive input spike trains that either increase or decrease their membrane potential. Through temporal accumulation, membrane potential may reach a specific firing threshold and neurons transmit information by firing discrete spike trains to neurons in the next layer. These characteristics emulate the information transmission and processing in the brain. It is therefore regarded as the next-generation neural network [103].

Since SNNs use non-differentiable spikes as information carrying agents, gradient-based backpropagation (BP) that uses gradients to optimize synaptic connections and neuron parameters in ANNs is not directly applicable in SNNs. Thus, one of the main challenges is to train and optimize the network parameters in SNNs. At present, the available methods for training SNNs can be divided into three categories: (1) unsupervised learning, (2) indirect supervised learning, (3) direct supervised learning.

In the first approach, weights are modulated to mimic synaptic interactions between biological neurons. A classic example is the spike time-dependent plasticity (STDP) [1-3]. However, due to the reliance on local neuronal activity rather than global supervision, STDP-based unsupervised algorithms have been limited to training shallow SNNs and can only produce low accuracy on complex datasets [4-6].

In the second approach, an ANN model is first trained and then converted to a SNN with the same network structure, where the firing rate of the SNN neuron is approximated as the analog

output of the ANN neuron. The ANN-to-SNN conversion has produced state-of-the-art (SOTA) performance in image recognition tasks [7].

The last approach is direct supervised learning, which uses a similar gradient descent technique used in ANNs to train SNNs directly. SpikeProp [128] was the first BP-based supervised learning method for SNNs that uses a linear approximation to overcome the SNNs' non-differentiable threshold-triggered firing mechanism. Further works include Tempotron [13], ReSuMe [14], and SPAN [15]. However, they could only be used for training single-layer SNNs. A surrogate gradient algorithm proposed by [86] introduces a differentiable surrogate function to approximate the derivative of spiking activity. It executes spatio-temporal BP in the training phase and is widely applied to train deep SNNs.

Although the ANN-to-SNN conversion and surrogate gradient-based algorithm can train deep SNNs, there are some limitations. For the ANN-to-SNN conversion, training an ANN model is fast. Nevertheless, the approach requires considerable inference time (from hundreds to thousands of time steps) to approximate the analog outputs [93, 95, 96, 129, 130], which leads to high memory consumption, larger latency and decreased energy efficiency, diminishing the benefits of SNNs [7-9]. For the surrogate gradient-based algorithm, although it is possible to train SNNs with arbitrary time steps, the fewer the time steps, the lower the accuracy of the trained model would be. Training high accuracy SNNs with this approach often requires many times more training time and computational resources than training ANNs.

In this chapter, we propose a novel SNN training method that combines the ANN-to-SNN conversion and direct training using the surrogate gradient. The method consists of two phrases: single-step SNN training and conversion to multi-step SNN. Specifically, during the training

phase, a single-step SNN augmented with Gaussian noise is trained by surrogate gradient-based BP and then converted losslessly to a multi-step SNN model to promote its generalization capability. Our training technique greatly reduces not only training and inference time but also achieves a high accuracy, which significantly improves the operating efficiency of SNN.

The following summarizes the primary contributions of this paper:

1) We propose a novel SNN training algorithm by introducing a noise distribution, which speeds up the training and inference time of SNN.

2) We compare our method's training and inference time with those of current methods. The experiments demonstrate that our method is 3-5 times faster for training than the surrogate-gradient based method, and more than 100 times faster than the ANN-to-SNN conversion for inference.

3) We argue that introducing noise in SNN has biological plausibility.

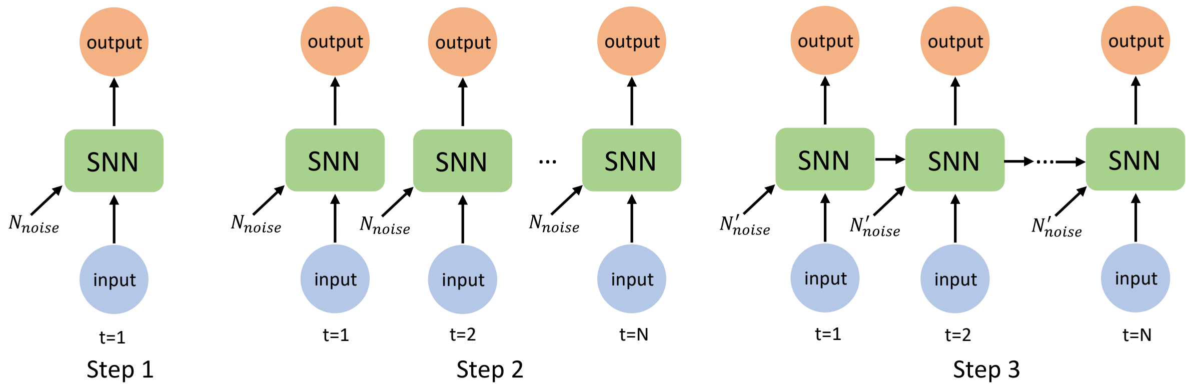


Figure 4.1 Three steps of our method to train a SNN model. Step 1, single-step SNN training with noise distribution N_{noise}. Step 2, copy N single-step SNNs and ensemble them together. N_{noise} varies over time-step t. Step 3, establish the temporal correlation among N different models.

4.2 Methods

4.2.1 Leaky Integrate-and-Fire model

The leaky Integrate-and-Fire (LIF) model is a fundamental unit in SNNs. It is a simplified representation of biological neurons that describes the non-linear relationship between input and output. The LIF neuron receives spikes over a specific period and it integrates them into its membrane potential, whose dynamics are governed by

$$H(t) = \lambda \cdot V(t-1) + \sum_i w_i \cdot S_i(t) \tag{4.1}$$

$$S(t) = \begin{cases} 1, & H(t) > V_{th} \\ 0, & H(t) \leq V_{th} \end{cases} \tag{4.2}$$

$$V(t) = H(t)\big(1 - S(t)\big) + V_{\text{reset}} \cdot S(t) \tag{4.3}$$

where $H(t)$and $V(t)$ represent the membrane potentials before and after triggering a spike at time t, respectively. λ represents the decay factor with a value of 0.5. $S(t)$ denotes the output of a neuron at time t, which equals 1 if there is a spike and 0 otherwise. $w_i \cdot S_i(t)$ is the weighted input of i-th neuron in the last layer at time step t. When the membrane potential of the LIF neuron reaches the firing threshold V_{th} (=1), the neuron fires one spike and the membrane potential is reset to the resting potential V_{reset} (here is 0).

4.2.2 Training single-step SNN and converting to multi-step SNN

We first train the single-step SNN with $T = 1$ and then convert it to a multi-step SNN with $T =$ N. When the total simulation step $T = 1$, the time dimension disappears and the network propagates forward only once. Consequently, the single-step SNN is actually an ANN with a Heaviside step function used as the activation function. Equation (4.1) is formulated as

$$H = \sum_i w_i \cdot S_i$$

(4.4)

Comparing Equations (4.1) and (4.4), the output of the multi-step SNN depends on both the input and the accumulated potential, while the output of a single-step SNN depends only on the input.

Due to the absence of the potential accumulation term in Equation (4.4) compared with Equation (4.1), we introduce a noise distribution N_{noise} representing the missing accumulated membrane potential during training the single-step SNN in order to perform the conversion into a multi-step SNN later, as Equation (4.5) shows. In particular, we assume that N_{noise} is a Gaussian distribution and distributes in each layer of the network independently. Thus, the dynamic of the neuron in a single-step SNN could be described by

$$H = N_{noise} + \sum_i w_i \cdot S_i$$

(4.5)

Our method consists of three following steps:

Step 1: Train the single-step SNN with Gaussian noise (see step 1 in Figure 4.1).

Step 2: Extend the temporal dimension from $T = 1$ to $T =$ N by directly modifying the value of T. This action means that we copy N single-step SNN and ensemble them together (see step 2 in Figure 4.1). For each individual, the inputs are the same. The average output of all individuals is the output of the ensemble model.

Step 3: Add the potential accumulation term using Equation (4.6) to establish the temporal correlation (see step 3 in Figure 4.1). The dynamic of SNN model after step 2 is formally different from the real SNN's dynamic because it lacks the process of potential accumulation

and the temporal correlation among different time step t. Consequently, we must add the item $\lambda \cdot V(t-1)$ to keep the formal consistency with Equation (4.1). We decompose N_{noise} into:

$$N_{noise} = N(\lambda \cdot V(t-1)) + N'_{noise} \tag{4.6}$$

N_{noise} could be represented as the addition of two Gaussian distribution items: the accumulated membrane potential distribution and the new noise distribution.

For the first item, we normalize $\lambda \cdot V(t-1)$ according to Equations (4.7) - (4.9) to approximate a Gaussian distribution ($\lambda \cdot V(t-1)$).

$$A = \lambda \cdot V(t-1) \tag{4.7}$$

$$\hat{A} = \frac{A - \mu}{\sigma} \tag{4.8}$$

$$N(\lambda \cdot V(t-1)) = \frac{\hat{A}}{\alpha \cdot max(abs(\hat{A}))} + \beta \tag{4.9}$$

Equation (4.8) converts the membrane potential distribution to a standard normal distribution approximately. μ and σ are the mean and standard deviation of A, respectively. Equation (4.9) guarantees that the potential distribution in each layer have the mean β and distributes between the interval $(-1/\alpha, 1/\alpha)$. By changing the values of α and β, we are able to change the mean and range of the distribution. ($max()$ and $abs()$ represent taking the maximum value and absolute value, respectively.)

For the second item N'_{noise}, we simply set it as a random Gaussian distribution. The mean and range of N'_{noise} also depends on α and β, because we need to ganrantee that the addition of two items in Equation (4.6) almost have the same mean and range as N_{noise} to avoid conversion loss.

The reason why we must introduce and keep the random noise distribution is that introducing the noise is equivalent to ensemble an infinite number of random models and helps promote accuracy during conversion to multi-step SNN. Because we introduce a random distribution when training single-step SNNs, the model generalizes well under different "assumed previous membrane potentials". As we increase the time step, the SNN with T simulation steps can be seen as a model consisting of T sets of models.

It is well known that a SNN model with more simulation steps T can increase performance. However, training a SNN with large T directly would increase not only the training and inference time but also the memory by T folds, so it is not very practical. The approach that we suggest can instantly construct a SNN model with large T with much less memory cost. The idea of introducing noise to generate an ensemble model was proposed in [70] and used for adversarial defense of ANN models. To the best of our knowledge, we are the first to use it in training and inference acceleration of SNNs.

4.3 Results and discussions

We conduct our experiments as Table 4.1 shows. SNNs are trained with MSE loss and Adam [91] optimizer. The initial learning rate is set to 1e-4. The cosine annealing warm restart [92]

learning rate schedule with $Tmax = 100$ adjusts the learning rate over training. Unless specified, all results are generated by default for $\alpha = 4$, $\beta = 0.5$, and N_{noise} in the range [0, 1].

Table 4.1 Network structures and training epoch for different datasets.

Dataset	Epoch	Network structure
MNIST	100	64C3-AP2-128C3-AP2-128C3-AP2-512FC-10FC
Fashion -MNIST	200	
CIFAR-10	1000	VGG-16
	800	ResNet-18

Note: nC3—Convolutional layer with n output channels, kernel size = 3 and stride = 1, AP2—2D average-pooling layer with kernel size = 2 and stride = 2, FC—Fully connected layer.

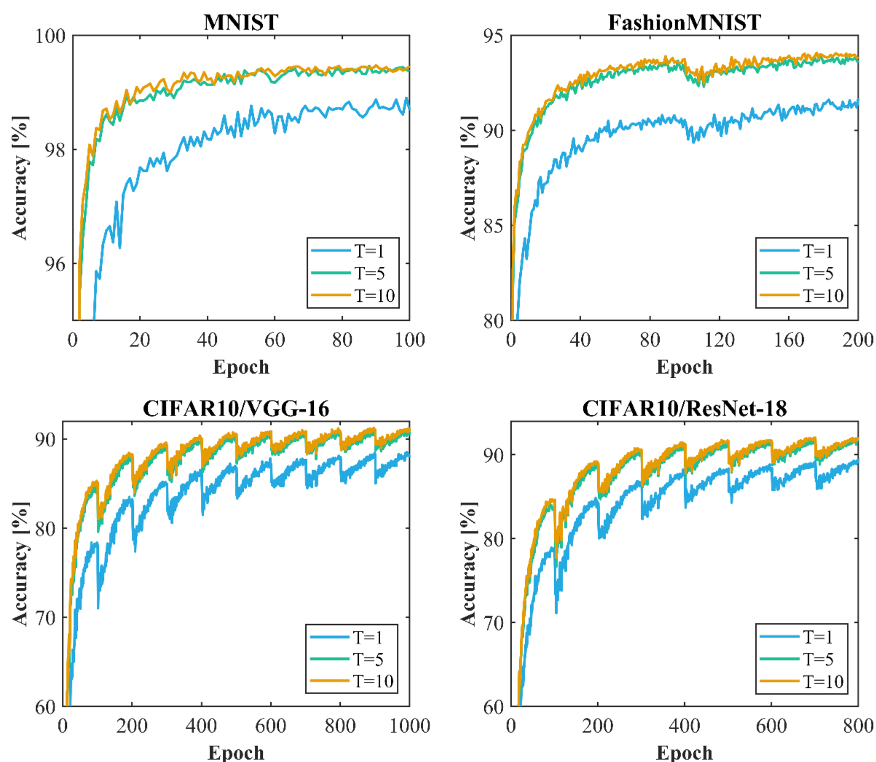


Figure 4.2 Inference accuracy of models on different datasets with T = 1, 5, 10 while training with N_{noise}.

4.3.1 Inference accuracy

In Figure 4.2, we plot the inference accuracy of single-step SNNs on different datasets and the inference accuracy of multi-step SNNs with total simulation step $T = 5$ and $T = 10$, respectively. It can be shown that as T increases, the accuracy of the SNN improves dramatically. The simulation step T could be directly converted to any values in real time.

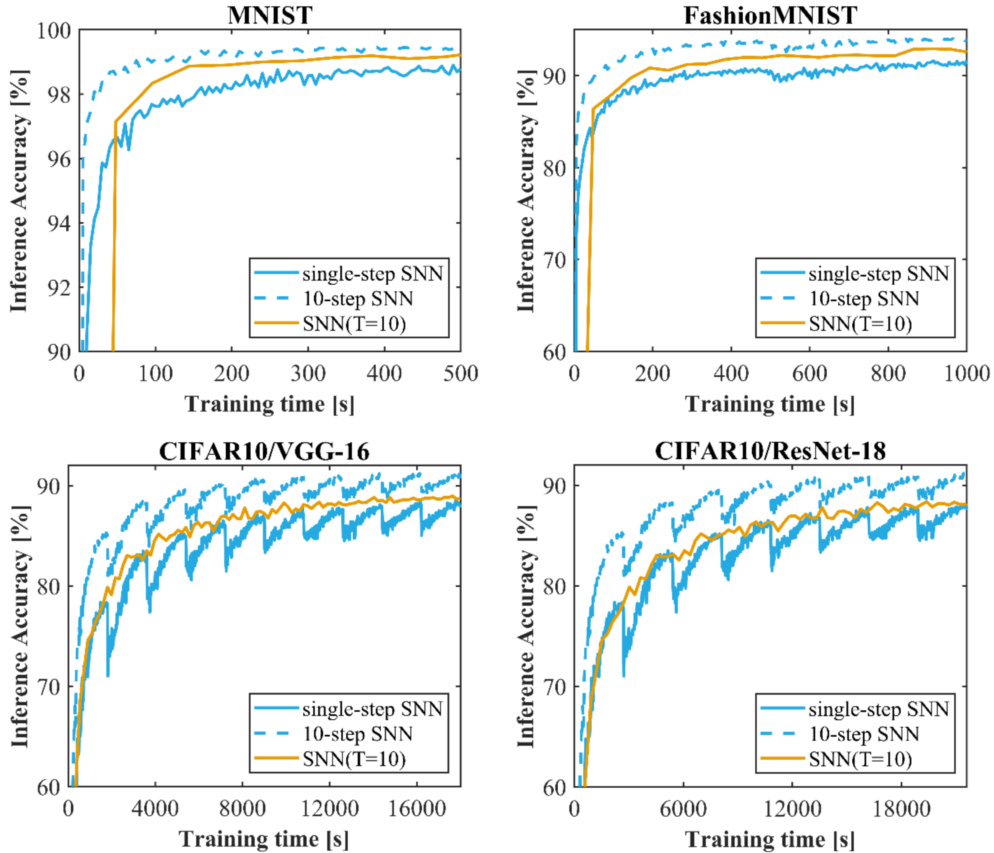


Figure 4.3 Training speed of SNNs when directly training an SNN(T=10) by the surrogate gradient approach versus training a 10-step SNN by our approach.

4.3.2 Comparison of training and inference time with related work

In Figure 4.3, we compare the training speed of SNNs when training an SNN(T=10) with a surrogate gradient versus training a single-step SNN and then extending to a 10-step SNN. Clearly, our method is substantially faster than directly training an SNN(T=10) using the surrogate gradient, and in the same amount of time, it achieves higher accuracy.

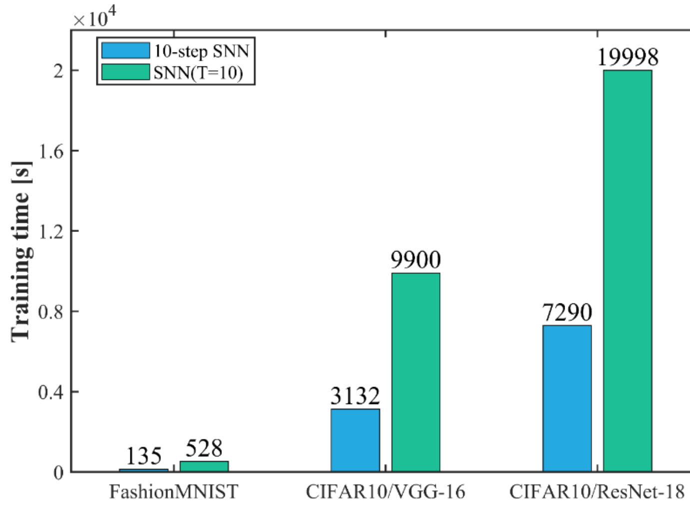


Figure 4.4 Training time of SNNs when directly training an SNN(T=10) by the surrogate gradient approach versus training a a 10-step SNN by our approach. The benchmark inference accuracy of the three models is 92%, 88%, and 90%, respectively.

To intuitively assess the difference in training time, we selected a benchmark inference accuracy for each model and halted training when the benchmark inference accuracy was achieved. The benchmark inference accuracy of the three models is 92%, 88%, and 90%, respectively. As shown in Figure 4.4, on FashionMNIST our method takes only 135s to reach the benchmark accuracy when SNN(T=10) takes 528s, which saves about 75% of the time. In CIFAR10/VGG-16, our method requires 3132s, whereas SNN(T=10) requires 9900s, a time savings of about 70%. In CIFAR10/ResNet-18, our model requires 7290s while SNN(T=10) requires 19998s, a time savings of approximately 65%. Also, with single-step SNN, we can choose a larger batch size and thus achieve faster parallel training. It is more convenient and feasible for groups that lack sufficient computational resources.

For inference time, we compare current advanced methods listed in Table 4.2 with our method. As demonstrated, the accuracy of extending to multi-step SNN (no more than 10 time steps) is able to attain an approximate accuracy of ANN-to-SNN conversion methods. In contrast,

ANN-to-SNN conversion requires hundreds to thousands of time steps, which is hundreds of times slower than our method. Compared with spike-based BP methods, our method also requires fewer time steps to reach close accuracy.

Table 4.2 Inference time comparison between our work and related work

Author	Method	Inference time	MNIST	Fashion MNIST	CIFAR10
[17]	Spike-based BP	12	-	-	90.53%
[131]	Spike-based BP	10	-	-	93.44%
[18]	Spike-based BP	20	99.50%	92.07%	93.5%
[9]	Spike-based BP	1	99.53%	-	84.67%
[129]	ANN-SNN	2500	-	-	91.46%
[4]	ANN-SNN	2048	-	-	91.36%
[132]	ANN-SNN	2500	-	-	91.89%
[16]	ANN-SNN	50/100	99.59%	-	90.95%
[96]	Hybrid	500	-	-	92.42%
[133]	Hybrid	200	-	-	92.02%
Ours	Hybrid	5	99.61%	93.89%	91.82%
Ours	Hybrid	10	99.64%	94.07%	92.07%

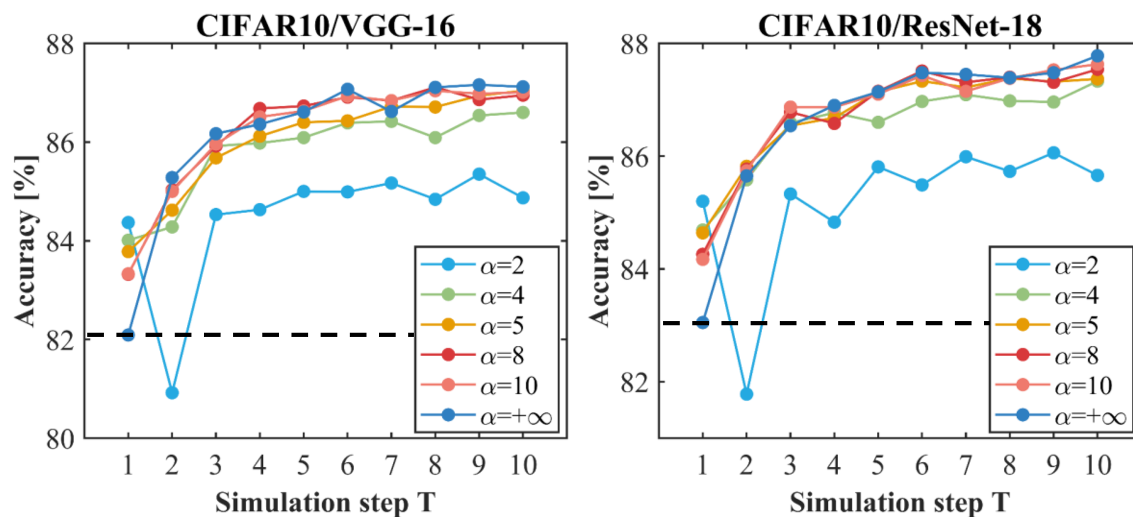


Figure 4.5 The impact of α on CIFAR10/VGG-16 and CIFAR10/ResNet-18. The black dotted line represents the accuracy of trained single-step SNN.

4.3.3 The impact of α

With Equations (4.6) and (4.9), we know that the parameter α controls the range of noise fluctuation N'_{noise}. Here, we attempt to alter the value of α to observe how the model's performance varies.

We plot the conversion accuracy for different values of α in Figure 4.5. When α is equal to 2, the potential range is [0,1], so the noise term N'_{noise} does not exist any more. It can be seen that there is a very slight improvement in accuracy with increasing time steps. In contrary, when N'_{noise} is present, the accuracy improvement is obvious and the different α values make models converge to close accuracy. These results indicate that noise plays the vital role in enhancing the accuracy of conversion.

When α is positive infinity, the range of noise N'_{noise} is [0, 1], membrane potential item disappears and the result equals to the outcome of step 2. We can see that the conversion from step 2 to step 3 has minor accuracy gap according to the figures.

4.3.4 The impact of noise type

In the previous sections, all of our experiments were performed by training the SNN model with Gaussian noise. In order to investigate whether it is only the uniform noise that brings the improvement of model accuracy, we replace the Gaussian noise with uniform noise during training to observe the effect of the noise type on models. As shown in Figure 4.6, we trained CIFAR-10 with uniform noise on both VGG-16 and ResNet-18 models. Models trained by uniform noise behave the same as with uniform noise, i.e., the accuracy is significantly improved when models are extended to multi-step SNNs. For single-step SNNs, models with Gaussian noise reach the higher accuracy than those with uniform noise. But after conversion, the gap is not obvious any more.

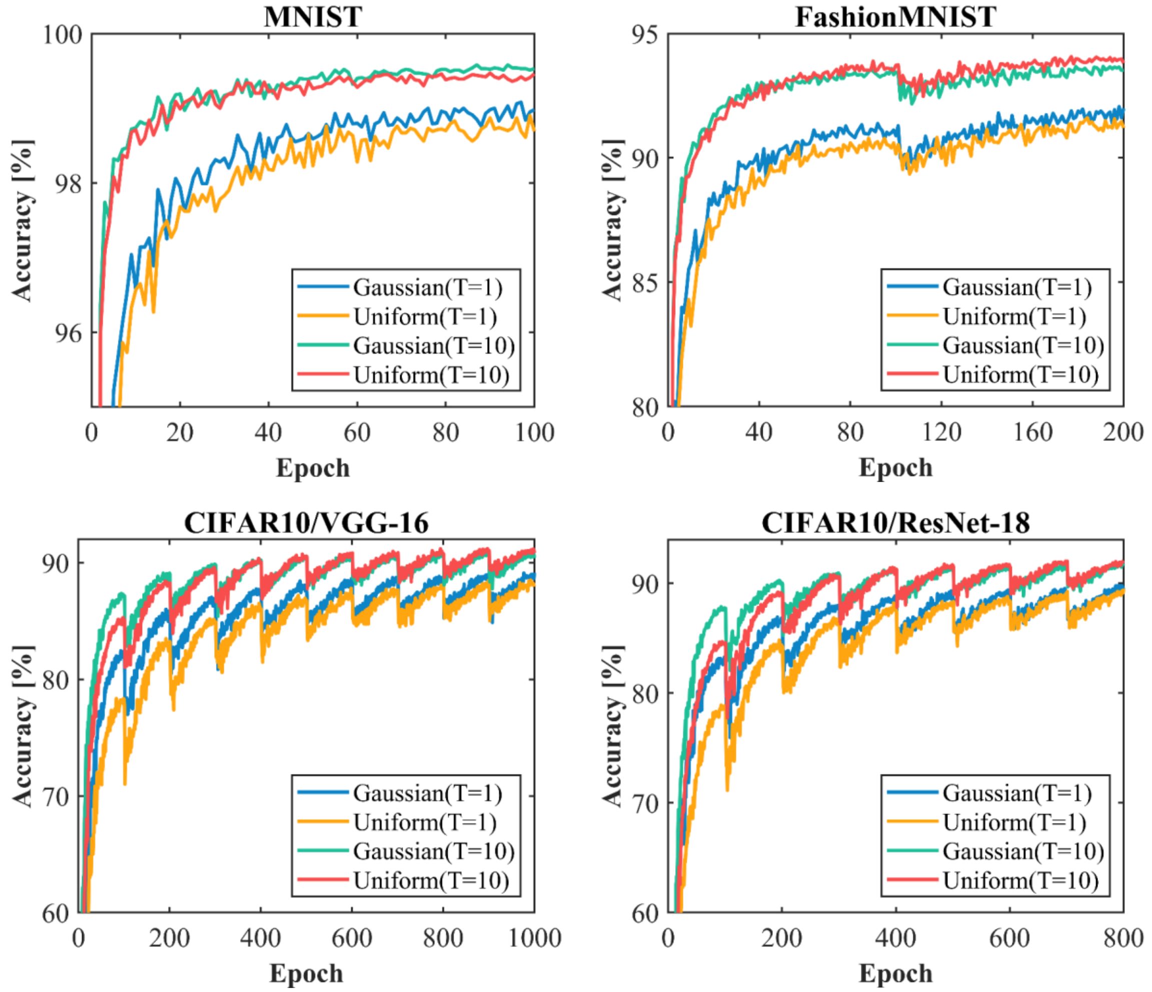


Figure 4.6 Inference accuracy of models on different datasets with T = 1, 10 while training using Gaussian noise and uniform noise, respectively.

4.3.5 Biological plausibility of uniform noise distribution in neuron

It is believed to be more biologically plausible when we keep some part of noise during conversion, as there exists lots of kinds of noise in biological neural system. We plot the dynamic of the spiking neuron in Figures 4.7(a) and (b). Figure 4.7(a) depicts the neural potential dynamic in the absence of input. The potential oscillates between the reset potential and the threshold, but no spikes fire. In Figure 4.7(b), when a neuron receives inputs, it begins to accumulate potential and fires spike. Such behavior is thought to be similar to the form of subthreshold neural oscillation mechanism in biological neurons. Neural oscillations are

rhythmic patterns of activity generated by the neurological system [123]. Many cognitive activities, including information transfer, perception, and memory, are believed to be related with neural oscillations. These oscillations are mostly caused by the interaction between individual neurons. Neural oscillation can emerge as oscillating membrane potentials or rhythmic action potentials in individual neurons. Subthreshold membrane potential oscillations are membrane oscillations that are below the firing threshold and hence cannot directly initiate action potentials. However, they can aid in sensory signal processing. As a result of subthreshold membrane potential oscillations, sensory systems, particularly for vision and smell, evolve. Subthreshold membrane potential oscillation (see Figure 4.7(c)) in the visual system helps process visual input and adjust to sensory input [134]. Additionally, oscillatory activity influences excitatory postsynaptic potentials, refining post-neural activities [135].

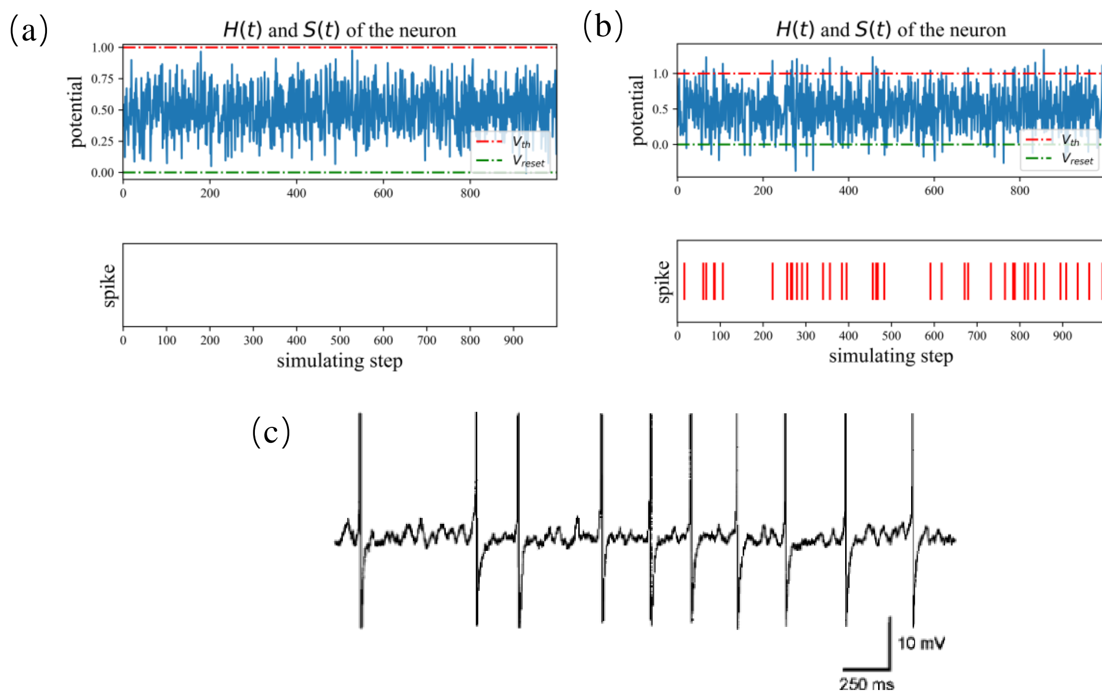


Figure 4.7 (a) Neural potential dynamic in the absence of input. (b) Neural potential dynamic when receiving input. (c) Subthreshold membrane potential oscillation. Source: Figure (c) is cited from [136].

4.4 Discussion and conclusion

We have used the proposed method to handle static vision problems in previous sections. For dynamic vison problems, such as videos' data, which include time series information, we should feed all the information into the network at once and use a 3D convolutional network rather than a 2D convolution network to deal with the input.

In this chapter, we propose a novel way of training SNNs that achieves accuracy improvement in multi-step SNNs by fitting the neural network to noise, which greatly spares the training and inference time of SNNs and allows fast training of SNNs with arbitrary simulation time compared to previous methods. Our approach combines the advantages of both direct training of SNN and ANN-to-SNN conversion. With a good balance of accuracy and training time, and a great saving of computational resources, this method can be used to train large SNNs quickly or SNN pre-training. The inclusion of noise is also proved to be more consistent with the dynamic mechanism of biological neurons. These points make our method promising for training deep SNNs in the future.

5 Spiking sampling network for image sparse representation and dynamic vision sensor data compression

Abstract

Sparse representation has attracted great attention because it can greatly save storage resources and find representative features of data in a low-dimensional space. As a result, it may be widely applied in engineering domains including feature extraction, compressed sensing, signal denoising, picture clustering, and dictionary learning, just to name a few. In this chapter, we propose a spiking sampling network. This network is composed of spiking neurons and it can dynamically decide which pixel points should be retained and which ones needs to be masked according to the input. Our experiments demonstrate that this approach enables better sparse representation of the original image and facilitates image reconstruction compared to random sampling. We thus use this approach for compressing massive data from the dynamic vision sensor, which greatly reduces the storage requirements for event data.

5.1 Introduction

Sparse signal representation has been demonstrated to be a highly effective technique for obtaining, representing, and compressing high-dimensional signals. Important signal classes, such as audio and images, have sparse representations with respect to a particular basis (e.g., Fourier and wavelet bases) or the concatenation of them. Furthermore, efficient and demonstrably successful techniques based on convex optimization or greedy pursuit are available for computing such high-fidelity representations [137].

Sparse representation is not only widely used in signal processing but is also useful for vision tasks. In the past few years, sparse representation has been applied in face recognition [138-144], image super-resolution [145], motion and data segmentation [146], denoising and painting [147-149], background modeling [150, 151], photometric stereo [152], and image classification [153, 154]. In almost all these applications, the use of sparse representation has achieved impressive results.

The capacity of sparse representations to reveal semantic information is influenced in part by a simple but crucial attribute of the data: despite the images' (or their features') naturally high dimensionality, images belonging to the same class demonstrate degenerate structure in many applications. In other words, they are situated on or close to low-dimensional subspaces, submanifolds, or stratifications. If a collection of representative samples is obtained for this low-dimensional distribution, we could anticipate that a typical sample will have a sparse (potentially learnt) representation over this basis. If appropriately computed, such a sparse representation might naturally encode semantic visual information [155].

In deep learning, many works have attempted to introduce sparse coding into neural networks. They usually mask certain input information randomly, which can be considered as a certain kind of sparse coding. BERT and GPT, for instance, are very effective pre-training techniques for NLP. In order to train models to anticipate the missing information, these techniques hold out a piece of the input sequence. There are tons of evidence that these techniques generalize very well and that the pre-trained representations perform admirably across a wide range of downstream tasks.

Methods exist for encoding masked images pick up representations from masked images that have been distorted. Convolutional networks are used by the Context Encoder [156] to fill in significant missing sections. iGPT [157] guesses unknown pixels based on pixel sequences. In the ViT study [158], masked patch prediction for unsupervised learning is considered. The most recent technique for predicting discrete tokens arises from BEiT. MAE [159] tries different masking methods to train the autoencoder which can be adopted to serve as the pre-training model. In most cases, the mask is a randomly generated sampling matrix. Recovering signals from fewer data gathered by a random measurement matrix is efficient. However, they constantly have issues with unclear quality of reconstruction [160].

Spiking neural networks (SNNs) are receiving increasing attention due to their low power consumption and bio-plausibility. Neurons in SNNs receive spike trains that either increase or decrease their membrane potential over time. When the membrane potential exceeds a certain threshold, the neuron fires one spike to next layer's neurons and reset its potential. These characteristics are similar to the way the brain transmits and processes information. It is therefore regarded as the next-generation neural network [103].

Since the spiking neural network (SNN) naturally outputs only 0 and 1 state values, we shall design a spiking autoencoder that generates a binary mask based on the input, where 0 means a certain pixel is not sampled and 1 means that the pixel is sampled at the input stage. Such a mask is multiplied with the input image to obtain the sampled image.

In this chapter, we propose a novel sampling network based on spiking neural networks, which is able to dynamically sample the input images, retain the valid pixels and remove the redundant pixels to output a sparse representation of the inputs. We validate its advantages over random sampling for network reconstruction on MNIST and CIFAR-10 datasets. Besides, we apply it to the compression of data generated by event cameras, which greatly reduces the space needed for data storage.

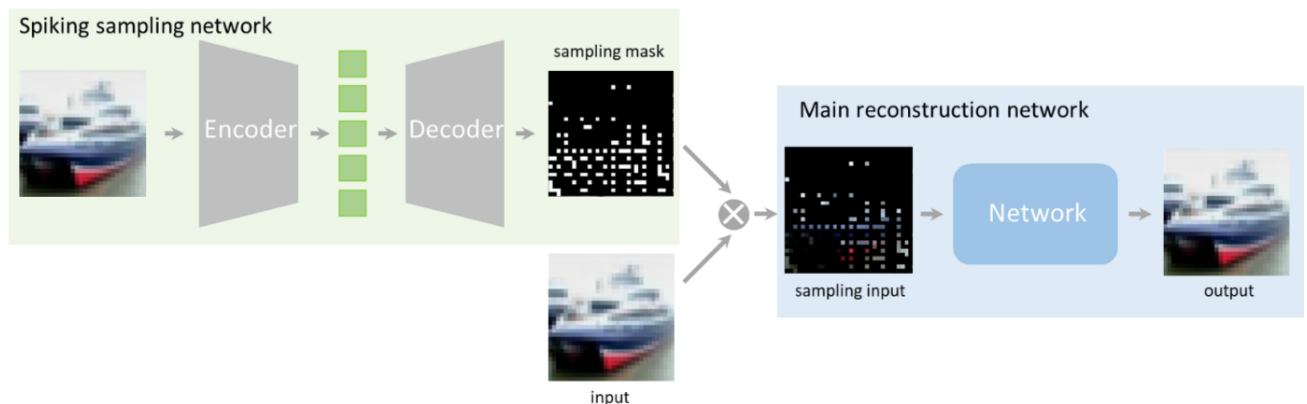


Figure 5.1 Architecture of the spiking sampling network. The output of spiking sampling network is a mask of the same size as the input.

5.2 Methods

5.2.1 Leaky Integrate-and-Fire (LIF) model

SNNs adopt spike trains as information carriers between neurons. Every spiking neuron in a SNN receives and emits spikes. The LIF neuron model is a popular bio-inspired simplified model for describing the dynamics of spiking neurons. The dynamics of the LIF model are

defined by [26].

$$H(t) = \lambda * V(t-1) + \sum_i w_i x_i(t)$$

(5.1)

$$S(t) = \begin{cases} 1, & H(t) > V_{th} \\ 0, & H(t) \leq V_{th} \end{cases}$$

(5.2)

$$V(t) = H(t)\big(1 - S(t)\big) + V_{reset} * S(t)$$

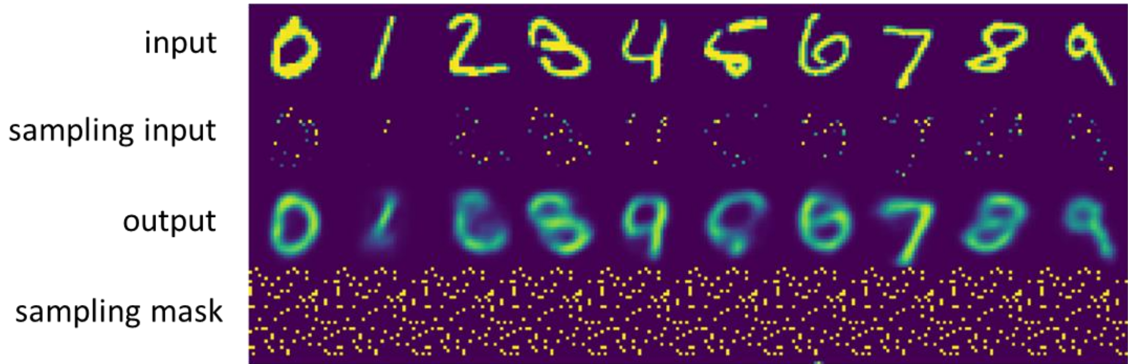
(5.3)

where $H(t)$ and $V(t)$ represent the membrane potentials before and after firing a spike at time t, respectively. V_{th} denotes the firing threshold, which is 1 in this paper. V_{reset} is the resting potential which is 0. $S(t)$ denotes the output of a neuron at time t, $w_i x_i(t)$ is the i-th weighted pre-synaptic input at time t, and λ is the decaying time constant with a value of 0.5.

5.2.2 Architecture and training of spiking sampling network

Figure 5.1 illustrates the architecture of the spiking sampling network. It is actually an autoencoder composed of spiking neurons. The neurons of all layers except the last layer have predefined thresholds V_{th} = 1. In the last layer, the threshold of the neurons is not a fixed value but varies dynamically with the input. The spiking neurons in the last layer only accumulate potentials over time and do not fire spikes until the last time step T. At instant T, we rank the accumulated potentials of all neurons from largest to smallest, and if we need to sample N pixel points, the Nth largest potential is used as the threshold V_{th} so that the number of neurons that fire spikes is N.

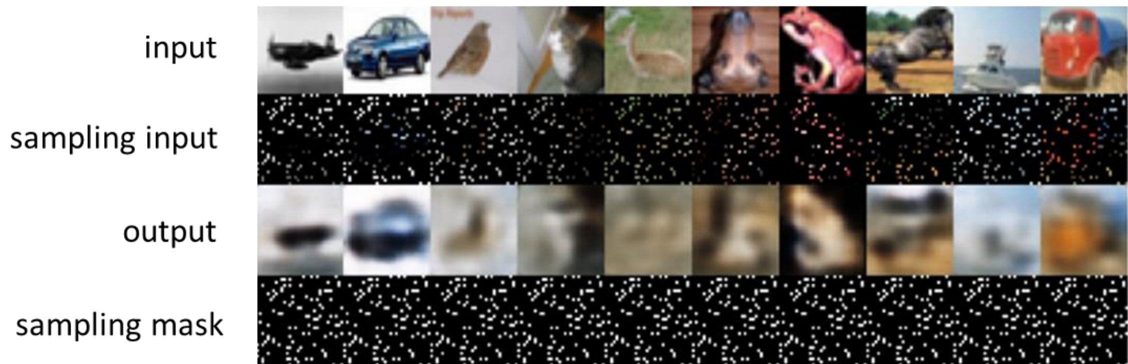
Random sampling



SNN sampling

(a)

Random sampling



SNN sampling

(b)

(c)

Figure 5.2 Comparison between random sampling and spiking sampling on (a) MNIST and (b) CIFAR-10. (c) Sampled pixels by spiking sampling network on CIFAR-10.

The output of the spiking sampling network is a sampling mask that has the same size as the input. This sampling mask will be multiplied by the actual input image to preserve the selected pixels. These pixels will be used as input to the main reconstruction network that is used for reconstructing the original image.

In contrast to the commonly used random sampling, our sampling scheme is implemented by a spiking neural network whose parameters can be optimized via back propagation. Thus, the network is able to automatically sample different pixel points for different inputs, depending on the main vision task.

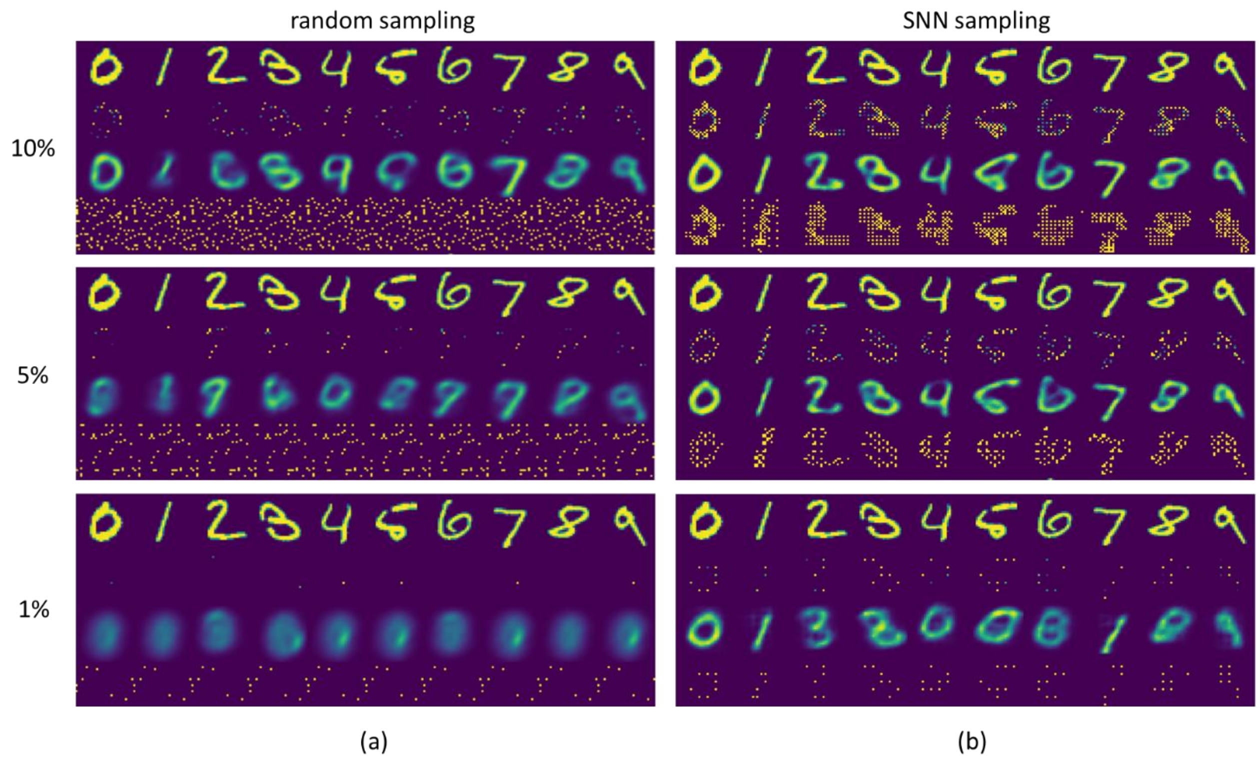


Figure 5.3 Different sampling rate comparison of random sampling and spiking sampling on MNIST

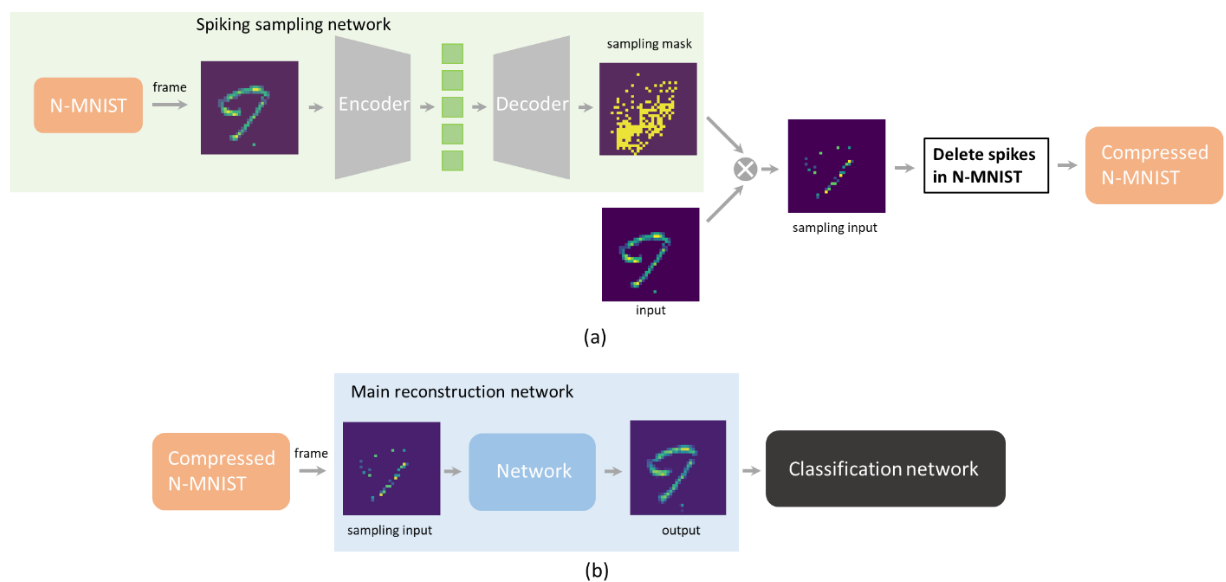


Figure 5.4 (a) Compression of N-MNIST dataset. (b) Classification validation of compressed N-MNIST dataset.

5.2.3 Data compression of dynamic vision sensor

Dynamic Vision Sensor (DVS), also called the event camera, is based on the principle of biosensing, which means that they report only the ON/OFF triggering of luminance in the observed scene [161]. Unlike conventional RGB cameras, which acquire raw data in a two-dimensional matrix, in event cameras, each pixel works independently and asynchronously, reporting changes in luminance as they happen or remain inactive while light intensity is constant [162]. In real-time interaction systems like robotics, drones, and autonomous driving, the DVS's distinctive features provide benefits over traditional vision sensors. In the near future, cloud and edge computing will be used to execute the majority of the services that do object/gesture recognition or classification. Therefore, in order to interpret visual data, these services would need to send spike events to cloud or edge servers [163]. Real-time transmission is also necessary in many circumstance. Despite the inherent compression offered by the neuromorphic sensing technology, further compression of the generated data may be advantageous for sending such data over Internet of Things (IoT), Internet of Things (IoV), and Industrial IoT (IIoT) situations [161]. Since the data storage and transmission bandwidth for onboard DVS processing and transmission are both limited, the compression of neuromorphic spikes is still a difficult problem that needs quick solves.

5.3 Experiments

5.3.1 Image reconstruction comparison

In Figure 5.2, we compare the effect of random and spiking sampling on image reconstruction at a sampling rate of 10%. We conduct the experiments on MNIST and CIFAT-10 datasets. The details of hyper-parameter selection and network architectures are listed in the appendix. We can see that for random sampling, the sampling positions are uniformly distributed over the entire image. The spiking sampling, on the other hand, changes the sampling positions, depending on the input. For the MNIST dataset, the spiking sampling focuses on sampling

over the figures while ignoring the surrounding background, and for the CIFAR-10 dataset, the sampling density is relatively small in the parts of the image with clean areas and increases in the areas with complex texture. For MNIST, spiking sampling reconstructs images more clearly than that of random sampling. For CIFAR-10, the color and shape of the reconstructed images using the spiking sampling network are more accurate than those from random sampling. Consequently, the pixels sampled by the spiking sampling are significantly more conducive to image reconstruction. Notably, the spiking sampling network does not tend to sample high pixel values, but it tends to sample more pixel points in regions where the pixel values vary drastically and allocate fewer sampling points in regions where the pixel values vary sparsely as the Figure (c) shows. This shows that the spiking sampling can effectively make the sparse representation of images.

Figure 5.3 shows the difference between reconstructed images with random sampling and SNN sampling at different sampling rates, respectively. It can be seen that random sampling at 10% sampling rate can no longer correctly distinguish all the reconstructed digits (e.g., digit 4), while using SNN sampling at 5% sampling rate can still clearly reconstruct all the images. At a sampling rate of 1%, random sampling is completely useless, while SNN sampling is still able to reconstruct some of the digits. Even with few sampling points, SNN sampling is still able to distribute the sampling points over the numbers to be reconstructed, effectively providing a sparse representation of the image. This indicates that the SNN network really learns the pixel points that are useful for reconstructing the image.

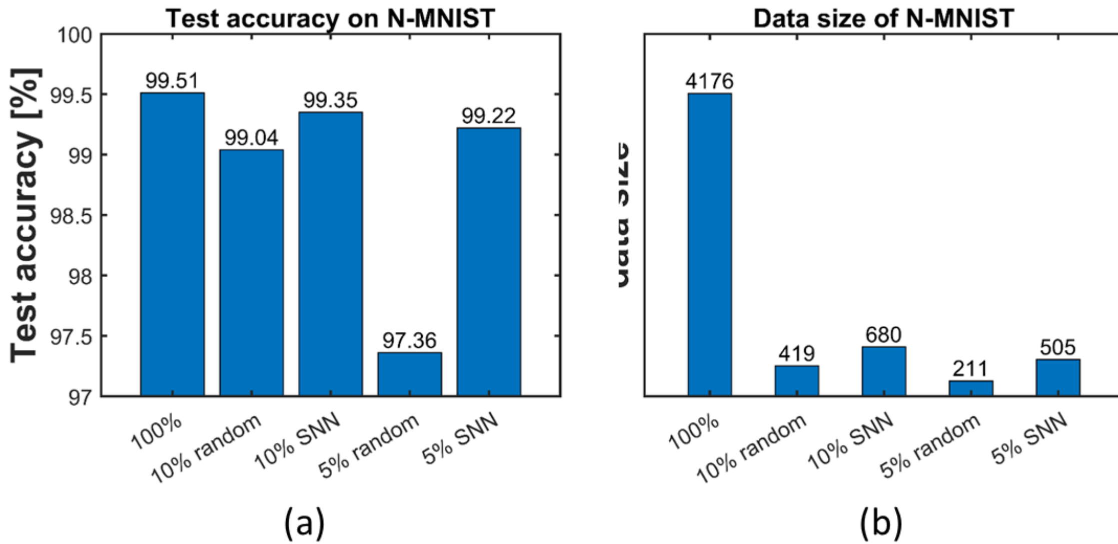


Figure 5.5 (a) Test classification accuracy on N-MNIST with different sampling method and rate. (b) data size comparison after different compressing rate by spiking sampling. The numbers on the bars represent the average number of spikes retained per sample for the dataset.

5.3.2 Event data compression

A spike event is composed of four basic elements, represented by a tuple {X, Y, t, p}: the spatial addresses X and Y, the timestamp t and the polarity p. The unique spike emission mechanism enables DVS to meet low bandwidth, low power, and low latency requirements. The unique spike emission mechanism enables DVS to meet the requirements of low bandwidth, low power consumption and low latency. At the same time, it also brings a huge amount of data. As an example, the commonly used handwritten numeric dataset MNIST only occupies about 11MB of storage space after compression, while the N-MNIST event dataset generated by MNIST still requires more than 1GB of storage space even after compression. Large datasets often require tens or even hundreds of GB of storage space, which puts a lot of storage pressure.

Since the output of DVS is very different from traditional frame-based image sequences, existing computer vision techniques cannot be directly applied to neuromorphic spike event

sequences. Integrating the original event stream into frame data is a common processing method. Therefore, we first render the spatio-temporal coordinates {X, Y, t} and polarity p of the neuromorphic sequences into frames before inputting them to the network. This rendering technique can be referenced in [26].

After training, we keep the spikes corresponding to the sampled pixels and remove the spikes corresponding to the unsampled pixels in the N-MNIST dataset based on the masks generated by the spiking sampling network (see Figure 5.4(a)).

To verify the validity of our retained event data, we do the classification task on the censored event dataset by a classification network (see Figure 5.4(b)), and the result is shown in Figure 5.5(a). It can be seen that when we use SNN sampling, the classification accuracy has only a slight accuracy loss at both 5% and 10% sampling rates, while the event data retained using random sampling causes a large accuracy loss on N-MNIST. Figure 5.5(b) shows the data size compared to the original N-MNIST dataset when it samples 5% and 10% by random sampling and spiking sampling network, respectively. Since spiking sampling focuses more on the spike-dense region in the image, it retains more spikes than random sampling at the same sampling rate, and the corresponding compression rate is somewhat smaller. Comparing Figures 5.5(a) and (b), the data size is reduced by 84% and 88% at a sampling rate of 10% and 5%, respectively, with a slight loss of accuracy, indicating that the spiking sampling network is able to sparsely represent the event dataset effectively.

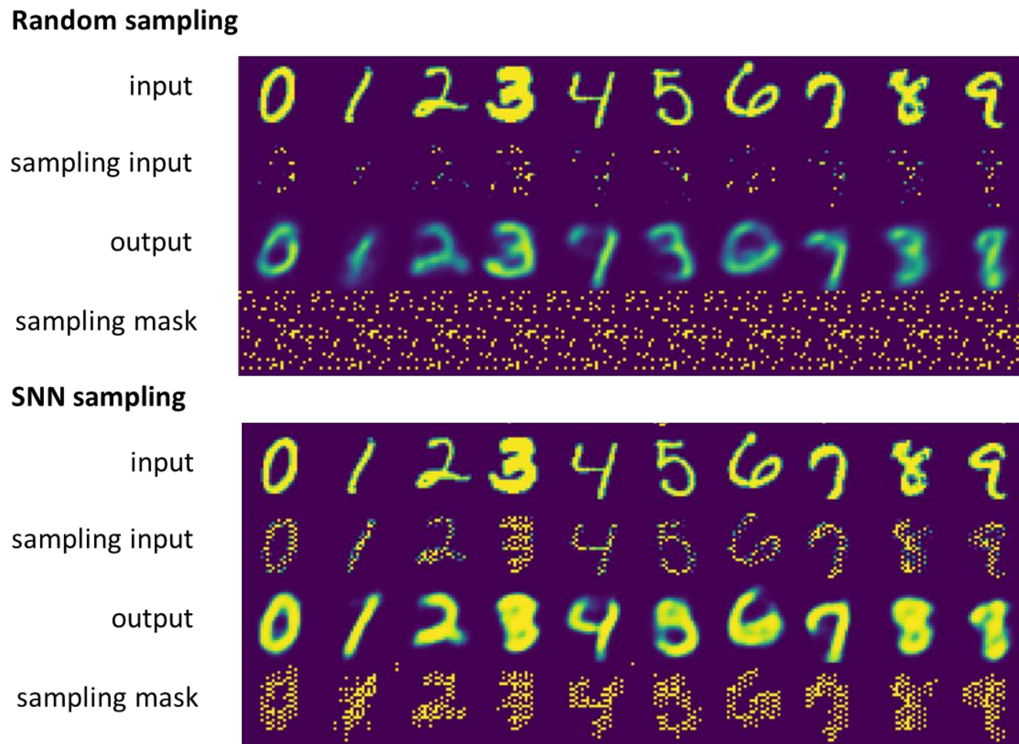


Figure 5.6 Reconstruction comparison of 10% random sampling and 10% spiking sampling on the main reconstruction network trained by random sampling.

5.3.3 Specificity and universality

In section 2.2, we know that image recovery needs to go through two steps, sampling and reconstruction. From the previous section, it can be verified that sampling method has a great impact on the reconstruction result. The main reconstruction networks obtained by taking different sampling methods for training also differ. In this section, we verify the sensitivity of the main reconstruction network to the sampling methods.

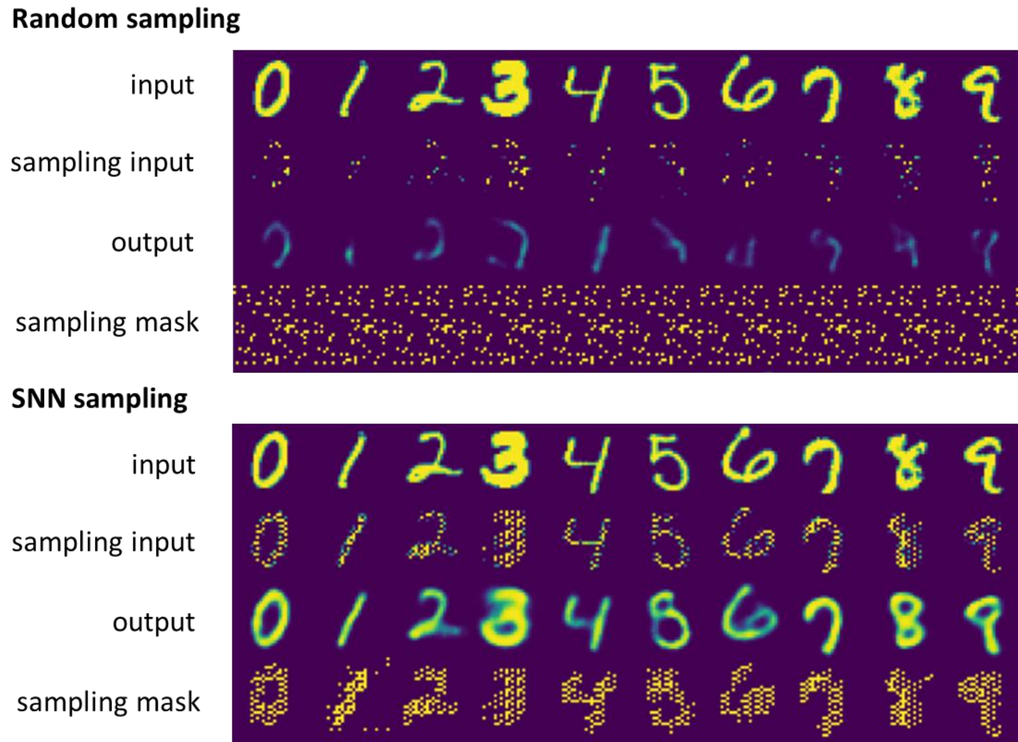


Figure 5.7 Reconstruction comparison of 10% random sampling and 10% spiking sampling on the main reconstruction network trained by spiking sampling.

After training, we are able to obtain two reconstruction networks, which target random sampling and spiking sampling reconstruction, respectively. Now we do both kinds of sampling separately and input the sampled pixels to the same reconstruction network to compare the reconstructed results. Figure 5.6 shows the output difference of main reconstruction network trained by random sampling, when we use random sampling and spiking sampling for test. We can see that even if we use random sampling during training, the quality of the image reconstructed by spiking sampling is no worse than random sampling during test. This shows that the main reconstruction network trained with random sampling has the good universality, and it is less sensitive to different sampling methods. Figure 5.7 shows the output difference of main reconstruction network trained by spiking sampling, when we use random sampling and spiking sampling for test. Random sampling has a great impact on this main reconstruction network, and the reconstructed images are poor. It

demonstrates that the main reconstruction network trained with spiking sampling is more susceptible to the influence of the sampling method and therefore it is more specific to the sampling method. Therefore this sampling method has more potential applications in terms of data privacy and security.

In summary, we conclude that spiking sampling enables higher reconstruction quality, but lead main reconstruction network to be specific with the sampling method; while random sampling makes the reconstruction process more difficult, but make main reconstruction network have better universality on the sampling method.

5.4 Discussion and conclusion

In this chapter, we propose a novel sparse representation method by a spiking sampling neural network. Different methods of compressing event data have also been proposed in some literatures [162-166]. The main differences between our method and these compression methods are 1) we directly use a SNN to sample the dataset. The end-to-end approach is much simpler; 2) we retain the pixel points of the original image without various linear and nonlinear transformations; 3) the compressed pixels are able to retain spatial information. We verify on static datasets that the network is able to learn sparse features of each sample independently by training. Compared to random sampling, the spiking sampling network performs better in image reconstruction. Our method can be applied to compress dynamic datasets with large amounts of data, which can greatly reduce the storage space and speed up data transfer.

5.5 Supplementary

Network and training details. Table 5.1 shows network structures for image reconstruction on

MNIST and CIFAR-10 and data compression on N-MNIST. Models are trained with MSE loss and Adam optimizer. The initial learning rate is set to 1e-4. SNN is trained by surrogate gradient [30]. The simulation time of SNN is 3 steps. For reconstruction, we trained models 100 epochs; for classification, we trained the model 20 epochs.

Table 5.1 Network structures for image reconstruction on *MNIST and CIFAR-10 and data compression on N-MNIST*

Dataset	Network Structure
MNIST	SNN: 16C3P1-MP2-4C3P1-MP2-16CT2S2-1CT2S2
	Main network: FC784-FC256-FC64-FC20-FC64-FC256-FC784
CIFAR-10	SNN: 16C3P1-MP2-4C3P1-MP2-16CT2S2-1CT2S2
	Main network: 12C4S2P1-24C4S2P1-48C4S2P1-96C4S2P1-48CT4S2P1-24CT4S2P1-12CT4S2P1-3CT4S2P1
N-MNIST	SNN*: 12C4S2-24C4S2P1-48C4S2P1-96C4S2P1-48CT4S2P1-24CT4S2P1-12CT4S2P1-2CT4S2
	Main network*: 64C3P1-64C3P1-64C3P1-64C3P1-2C3P1
	Classification network: 128C3-128C3-MP2-FC2048-FC100-FC10

Note: nCm—Convolutional layer with n output channels, kernel size = m and stride = 1, nCm—transposed convolutional layer with n output channels, kernel size = m and stride = 1, MP2—2D max-pooling layer with kernel size = 2 and stride = 2, FC—FC layer. * represents all convolutional layers are 3D layers.

6 A Spiking Neural Network with Spike-timing-dependent Plasticity for Surface Roughness Analysis

Abstract

Spiking neural network (SNN) utilizes spike trains for information processing among neurons, which is more biologically plausible and widely regarded as the third-generation artificial neural network (ANN). It has the potential for effectively processing spatial-temporal information and has the characteristics of lower power consumption and smaller calculation load compared with conventional ANNs. In this work, we demonstrate the feasibility of applying SNN to classify tactile signals collected by a bionic artificial fingertip that touches a group of real-world metal surfaces with different roughness levels. A two-layer SNN is adopted and trained using an unsupervised learning method with spike-timing-dependent plasticity (STDP). Experiments show that the trained SNN can categorize the input tactile signals into different surface roughness of metal textures with more than 80% accuracy. This work lays the foundation of applying SNNs to more complex tactile signal processing in robotics, manufacturing, and other engineering fields.

6.1 Introduction

Surface roughness is one important object property closely related to wear resistance, fatigue strength, vibration and has an important impact on the service life and reliability of mechanical products [167]. Tremendous efforts have been made to recognize the surface texture using artificial tactile sensors or artificial fingers [168-171]. For example, reference [169] developed a silicon MEMS-based capacitive tactile sensor array to differentiate between surface textures, including polycotton and nylon. Reference [172] fabricated a 2 × 2 array of four microelectromechanical systems (MEMS) tactile microsensors based on microfabrication technology, which was embedded in a polymeric packaging with fingerprint-like structures. In recent years, a growing body of literature discriminates different surfaces by tactile sensors combined with machine learning[173-175]. Support vector machines (SVM) and k-nearest neighbors (kNN) were applied to classify surface roughness through extracting a series of features of samples[174]. In the literature [173], multi-sensor fusion was incorporated with machine learning to recognize surface roughness.

With the development of artificial intelligence, deep learning, especially neural networks are widely studied recently. SNN is a special class of ANN, where neurons communicate by spike train[176]. It is considered as the third generation of ANNs [177] because SNN is generally based on more biologically plausible neuronal models, i.e., more capable of capturing the complex temporal dynamics just like biological neurons [178]. Potential advantages, e.g., energy-efficient and less delay, occur compared with a conventional neural network such as Convolutional Neural Network (CNN) due to its event-based triggered property. In addition, SNN also demonstrated the ability to capture the time correlation between time variables in streaming data. Thus, a considerable amount of research on SNNs has been explored recently,

especially attempts have been made to deal with classification problems using SNNs [132, 179-182].

Several attempts have been made to discriminate rough surfaces by a biomimetic fingertip with piezoelectric sensors [173-175]. In the previous work [175], analog tactile signals generated from polyvinylidene difluoride (PVDF) films are fed as input to the Izhikevich neurons to obtain spike trains, and two distinct decoding schemes based on k-nearest neighbors (kNN) are used for surface roughness discrimination. However, this method distinguishes surfaces with only 77.6% classification accuracy. It is possible to get higher surface roughness discrimination accuracy (with an overall 80% accuracy) only for rougher surfaces ($Ra > 1$ μm) [174]. Sensor fusion including piezoelectric sensors and optical sensors [173] can extract more information from sampled signals and has a better ability to distinguish smoother surfaces ($Ra < 1$ μm). However, all these approaches are not simple and fast enough due to the complicated process before classification.

This study makes a major contribution by building a simple and fast end-to-end SNN to discriminate surface roughness. Particularly, we firstly transfer sampled electric signals into spike train as inputs of two layers SNN; then, an unsupervised method is applied to update synaptic weights according to the firing rate between pre- and post-synaptic neurons; finally, we assign a class to each neuron according to the response of inputs. The remaining part of this chapter proceeds as follows: Section 6.2 presents methods for the combination of tactile sensor signals and SNN. In Section 6.3, we evaluate the proposed method on surface roughness discrimination and compare our method with previously proposed methods. Finally, we conclude with a further discussion in Section 6.4.

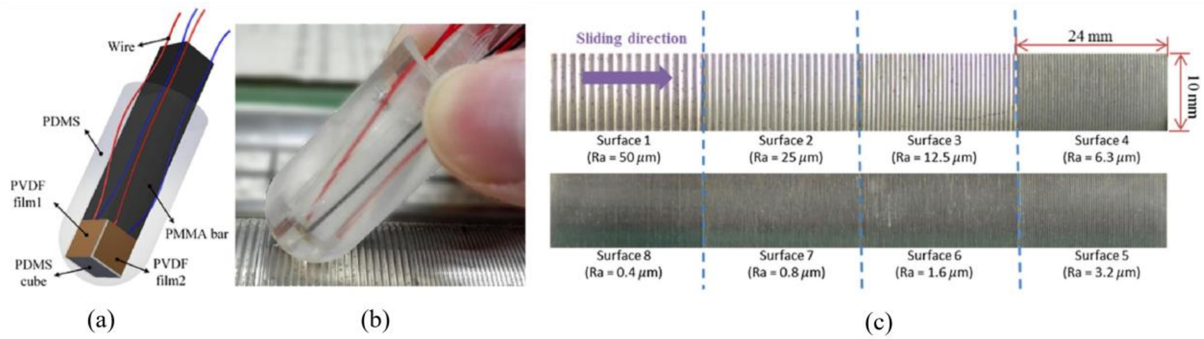


Figure 6.1 (a) The structure of designed biomimetic artificial fingertip. (b) Biomimetic fingertip sliding along the test surface. (c) Eight solid nickel test surfaces with different roughness values.

6.2 Methods

6.2.1 Experimental Setup

Due to the piezoelectric effect, one piezoelectric sensor is sensitive to mechanical force changes and could generate electrical signals that are proportional to the mechanical deformation of an object [183]. A bio-inspired tactile piezoelectric sensor was used to collect tactile datasets for each class of surface roughness.

The tactile sensor was designed by mimicking FA-I type mechanoreceptors in human fingertips which are extremely sensitive to dynamic stimuli and vibrations [174, 184]. It includes two commercial 28 mm thick PVDF films that were cut into the size of 4 mm × 4 mm. The PVDF film is a class and common material for manufacturing tactile signals. It exhibits a high-frequency response when sliding and is able to measure the lowest frequency about 0.01 Hz based on its property of piezoelectric effect, thus making them highly suitable for measuring vibrations. They are perpendicular to each other, and both were glued on the top of a polydimethylsiloxane (PDMS) cube. The PDMS cube was connected to one tip of a 4 mm × 4 mm × 45 mm polymethyl methacrylate (PMMA) bar. The PMMA bar with soft cured PDMS cube and PVDF films sensor was covered by a layer of PDMS (Figure 6.1(a)). The size of our

tactile sensor is comparable to human fingertips. Compared with human fingertips, the PMMA bar, PVDF films, and PDMS layer function as bone, mechanoreceptors, and skin, respectively [174].

We collected the piezoelectric signals on the test samples. The test samples consist of eight solid nickel surfaces with roughness values (Ra) of 50 μm, 25 μm, 12.5 μm, 6.3 μm, 3.2 μm, 1.6 μm, 0.8 μm, 0.4 μm (Figure 6.1(c)). The biomimetic fingertip was controlled to slide across different surfaces and generate vibratory stimuli. To simplify the operation and experimental setup, the sliding process was manually controlled at a speed of about 0.2 m/s. The discrimination result is expected to be further enhanced if the sliding speed and grip strength are controlled more precisely by devices like a robotic arm. The bio-inspired tactile sensor slid ten times on the surface of each test sample, which generated 80 time-series samples in total for each PVDF film. Analog outputs from the PVDF films are amplified via a custom amplifier and digitalized via DAQCard (USB-6225, National Instruments, USA) [184].

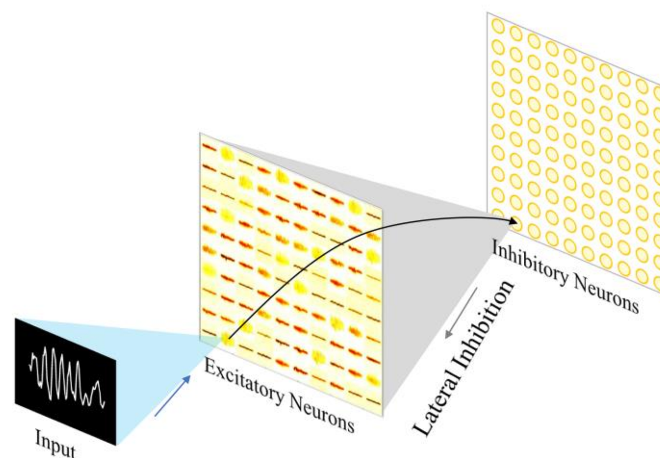


Figure 6.2 Illustration of SNN network structure for tactile signal processing.

6.2.2 SNN architecture

We adopt a two-layer feedforward SNN for tactile signal classification. It consists of one input layer and one output layer (see Figure 6.2). The SNN was constructed, referring to the structure proposed previously [181]. Input patterns were coded as Poisson spike processes, and the firing rates are proportional to the intensities of the corresponding pixels in the images. Each Poisson spike train is fed to the excitatory neurons of the output layer with all-to-all connections. The output layer consists of the excitatory neurons and the inhibitory neurons. The connection from the excitatory neurons to inhibitory neurons is in a one-to-one fashion, i.e., each of the excitatory neurons is connected to one corresponding inhibitory neuron at the same position. At the same time, each inhibitory neuron inhibits all excitatory neurons except for the one from which it receives an input.

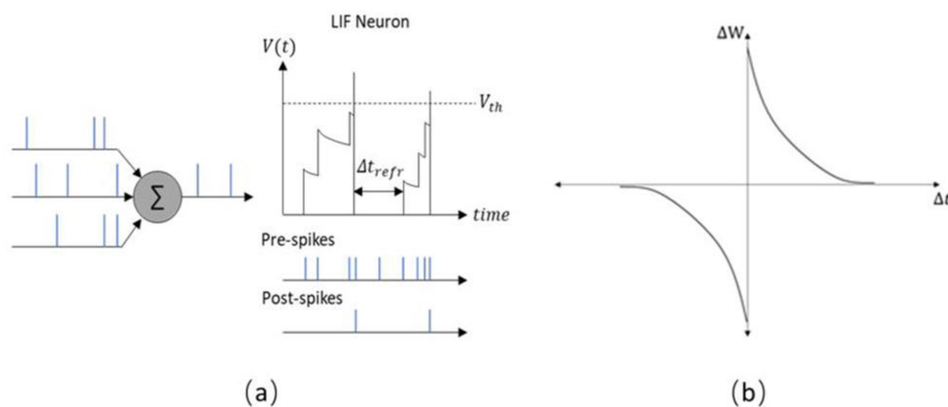


Figure 6.3 (a) LIF neuron model. (b) Schematic of the classic STDP.

6.2.3 Neuron and synapse model

For the dynamic neuron model, we chose the leaky integrate-and-fire (LIF) model, which was proposed based on the simplified model of biological neurons and widely used by SNN. The dynamic membrane potential u in this model is described by Equations (6.1) - (6.5).

$$\tau \frac{du}{dt} = u_{rest} - u + I, \qquad u < V_{th}$$

(6.1)

$$I = g_e(u_e - u) + g_i(u_i - u)$$

(6.2)

$$\tau_{g_e}\frac{dg_e}{dt} = -g_e + \sum_{i}^{n}\sum_{k} w_i^e\,\delta(t - t_i^k)$$

(6.3)

$$\tau_{g_i}\frac{dg_i}{dt} = -g_i + \sum_{j}^{m}\sum_{k} w_j^i\,\delta(t - t_j^k)$$

(6.4)

$$\text{fire a spike \& } u = u_{reset}, \quad u \geq V_{th}$$

(6.5)

where τ, τ_{g_e} and τ_{g_i} are time constants, u and u_{rest} are the membrane potential and resting membrane potential, respectively. u_e and u_i are the equilibrium potentials of excitatory and inhibitory synapses. I is the total pre-synaptic input, n and m are the numbers of excitatory and inhibitory synapses, g_e and g_i are the excitatory and inhibitory conductance. w_i^e and w_j^i are the excitatory and inhibitory connection weights, respectively. δ is the pre-synaptic input that equals 1 at the moment of firing a spike; otherwise, it is 0. u_{reset} is the reset membrane potential once u exceeds a given potential threshold V_{th}. All parameters chosen are set are within bio-plausible ranges.

As shown in Figure 6.3(a), when a neuron receives pre-synaptic spikes, it will accumulate the membrane potential according to Equation (6.1); once the membrane potential of the neuron exceeds its membrane threshold V_{th}, the neuron will fire a spike and immediately restore the initial potential u_{reset}. The neuron will be in a refractory period for the next few milliseconds,

which means the membrane potential will not change even if it receives spikes over this time. In this model, the firing threshold changes dynamically due to a dynamic threshold method adopted. The dynamic threshold is a bio-plausible feature initially discovered in the neural system [185-189]. The membrane threshold of each neuron depends on not only v_{th} but also an extra variable θ which slightly increases then exponentially decreases to the original value every time a neuron fires, which is described by Equations (6.6) - (6.7). The threshold of a neuron will be higher with more spikes fired, and in turn, more input is necessary in order to let the neuron spike. The purpose is to prevent a single neuron from firing too many spikes, thereby dominating the results in the output layer.

$$V_{th} = V_{th} + \theta \tag{6.6}$$

$$\tau_{\theta} \frac{d\theta}{dt} = -\theta \tag{6.7}$$

where τ_{θ} is the time constant of θ.

Synaptic weights from input neurons to excitatory neurons were updated using Spike-timing-dependent plasticity (STDP). STDP is a widely used unsupervised learning algorithm in SNN. According to the Hebbian learning rule, the strength of the synaptic connection between two neurons should be increased or decreased in proportion to the product of pre-synaptic and post-synaptic neuron activation[190]. STDP is considered an extension of Hebbian's theory. Under the STDP process, the activity between two neurons, if the information of other neurons is received before its activity, the connection between the two neurons will be strengthened. Conversely, if the neuron itself becomes active before receiving information from other neurons, the connection between the two neurons will weaken.

The increment of weights ΔW can be expressed by Equation (6.8) according to the model of STDP [191, 192].

$$\Delta \mathrm{W} = \begin{cases} A_{+} \exp\left(\frac{-\Delta t}{\tau_{+}}\right), & \Delta t \geq 0 \\ -A_{-} \exp\left(\frac{\Delta t}{\tau_{-}}\right), & \Delta t < 0 \end{cases}$$

(6.8)

where A_{+} and A_{-} are learning rates, Δt is the time difference between pre- and post-synaptic spikes, and τ_{+} and τ_{-} are the time constants of the positive and negative time difference, respectively.

Figure 6.3(b) shows the synaptic changes. In addition to the change of synaptic strength, we also use the synaptic scaling mechanism to increase the competition among synapses between the input and excitatory layers. Synaptic scaling is a homeostatic plasticity mechanism observed in many experiments, especially in visual systems and the neocortex [193]. The synaptic scaling normalizes the synaptic weights through Equation (6.9) after each sample is trained.

$$w_N = \beta \frac{w}{\sum w} N_{pre}$$

(6.9)

where N_{pre} is the number of synapses connected to a single target neuron, and β is a scaling constant which was set to 1 [193].

It should be noted that STDP and synaptic scaling were only used for the connection between input and excitatory layers. All weights from the excitatory layer to the inhibitory layer were initialized to 10.4, and weights from the inhibitory layer to the excitatory layer were 17 [1]. These settings guarantee that inhibitory neurons can be triggered as long as they receive one spike. The excitatory-inhibitory weights would not change after initializing. For the values of the above variables, see Table 6.3 in the appendix.

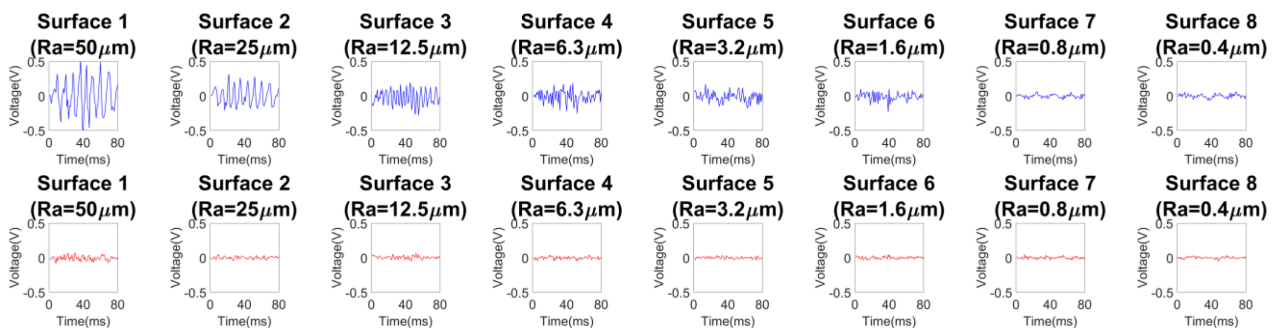


Figure 6.4 Typical tactile signals generated by two perpendicular PVDF films when sliding on eight surfaces with different roughness values. PVDF1 is perpendicular to the sliding direction, while PVDF2 is parallel to the sliding direction.

6.2.4 Dataset

When sliding laterally on eight test surfaces with different roughness values, the sample tactile signals generated by two PVDF films are shown in Figure 6.4. Each dataset includes 8 classes, and each class has 10 samples. For every class, the 10 samples were randomly split into 7 samples as the training set, and the remained 3 samples as the testing set. In order to enhance the generalization ability of the model and avoid overfitting, we increase the number of training samples through data fragmentation. Specifically, for each class, we include samples 1-4 into group 1 and samples 5-8 into group 2. We firstly split each sample into four pieces of equal length (Figure 6.5). In this way, each sample in group 1 would have four pieces, while each sample in group 2 has three pieces. Then, for each time period, we pick one piece from a sample and combine them together. In total, for each class, we obtain 337 training samples. As a whole,

the training set has 2696 samples, and the testing set remains to include 24 original samples. The split and combination of data are shown in Figure 6.5.

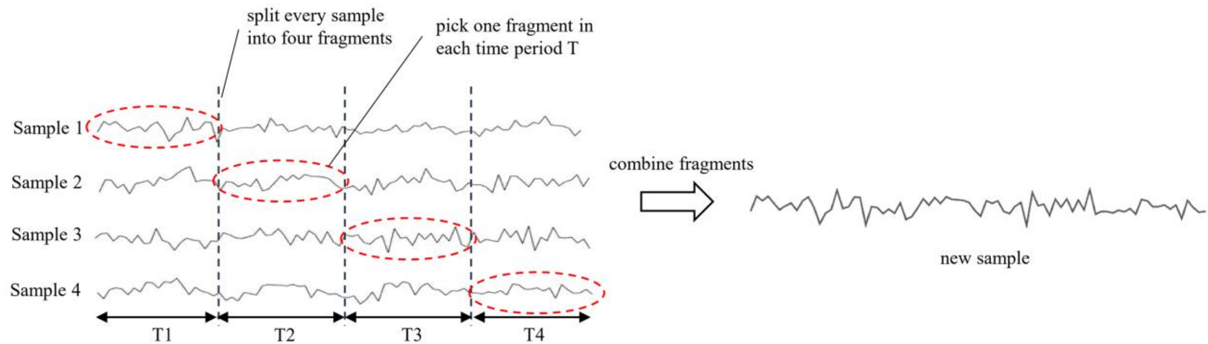


Figure 6.5 The split and combination of raw data.

6.2.5 Input Encoding

SNN exhibits the natural ability of spatiotemporal coding of input and thus holds the potential advantage of efficient coding through sparse activities, particularly for continuous spatiotemporal inputs. In the brain, it turns out that most of the information is encoded by the number of spikes in a short window [194]. Reference [195] demonstrates that spikes contain features of roughness and contribute to a firing rate code. So we encode the sampled tactile signals by rate coding.

Firstly, We represent each sample as a 28×28 grayscale image. Therefore, there are a total of 2696 training images and 24 test images. Then each image is coded as a Poisson spike train. The Poisson spike train is generated using Equations (6.10)-(6.12) [196].

$$P\{1\ spike\ during\ \delta t\} = r * \ \delta t \tag{6.10}$$

$$r = 0.25 * intensity$$

(6.11)

$$N = r * T$$

(6.12)

Equation (6.10) describes the possibility of producing a spike during every time step. In the paper, each time step in the simulation has δt = 1 millisecond. The firing rate r in (6.11) is proportional to the intensity of the corresponding pixel on the image. Equation (6.12) derives the average number of spikes N generated by one pixel, which is equal to the product of r and the simulation time T. This conversion process may cause information loss. As mentioned in the paper [107], the longer the simulation time is, the less loss from the original static image to a rated-coded spike image the conversion loss is, which can also be seen in Figure 6.6. Different simulation time presents varying degrees of difference between an original image and its rate-coded spike train. However, a long simulation time will cause a very long training process and more computation consumption. For our process, the simulation time was set as 350 ms in order to trade off training time and loss. The rate of generated spike train is initialized between 0 and 63.75 Hz according to (11) since the maximum intensity of a pixel is 255. Specifically, while the excitatory neuron layer generates less than 5 spikes, the input firing rate increases by 32 Hz and is presented again for 350 ms until no less than 5 spikes are emitted. It aims to guarantee that spikes can propagate to the deeper layer.

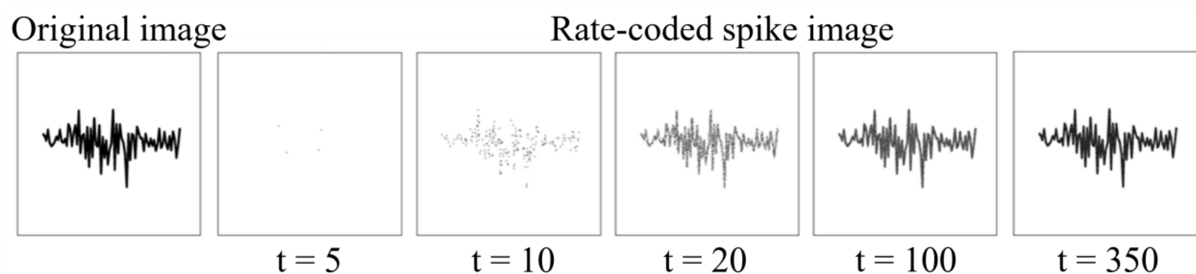


Figure 6.6 An original static image is encoded into a spike map over various time steps using rate coding.

6.2.6 Training and classification

The presentation time of each input sample takes 350 ms for each sample. It is followed by a resting time of 150 ms to restore all variables of all neurons to their resting values except the weight and adaptive threshold. For the testing process, we set the learning rate to zero and used the SNN with trained weights and thresholds. We also assigned a label to each excitatory neuron in the output layer, which depends on its highest average numbers of fired spikes to all eight classes of surfaces over one presentation of the training set. Then the classification accuracy of the SNN on the test set is measured based on averaging the responses of each neuron per class and choosing the class with the highest average firing rate [181]. The number of neurons in the excitatory and inhibitory layer was set as 400. We trained the model with 1 to 15 epochs of the training dataset.

6.3 Results

Figure 6.7 (a) shows the test result comparison between the augmented PVDF1 dataset and the augmented PVDF2 dataset. It reveals that there has been a steady accuracy after one epoch training and the trend increases slowly with more training epochs for both PVDF1 and PVDF2. After 15 iterative training, the test accuracy of the model on the PVDF1 dataset achieves 83.3%, in contrast with 54.17% on the PVDF2 dataset. Throughout the whole iteration process, the test performance of PVDF1 has been higher than that of PVDF2. This result suggests that SNN can better extract features to distinguish different categories with the PVDF1 dataset compared with the PVDF2 dataset.

Figure 6.7 (b) compares the trend of test accuracy between the augmented PVDF1 dataset and the original PVDF1 dataset with the increasing number of training examples. There is a significant promotion after data augmentation. It states that splitting and combining segments

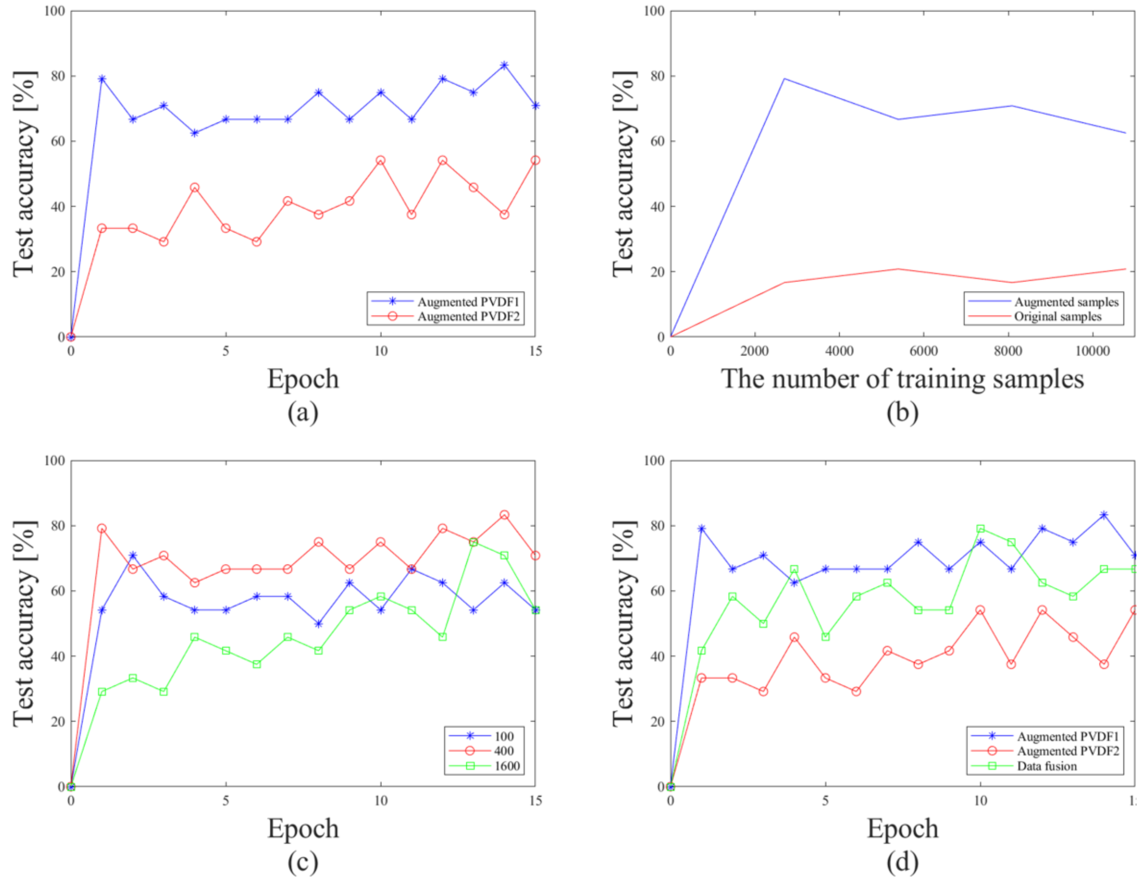


Figure 6.7 (a) Test accuracy comparison between augmented PVDF1 and augmented PVDF2 datasets with 15 training epochs. (b) Test accuracy comparison between augmented PVDF1 dataset and original PVDF1 dataset. (c) Test accuracy comparison among the different numbers of excitatory neurons in the output layer. (d) Test accuracy comparison among fusing both augmented PVDF data and single augmented PVDF data with 15 training epochs.

of signals effectively decreases the overfitting of SNN and promotes network generalization capability.

In addition to the 400 neurons model, we also tested models with 100 and 1600 excitatory and inhibitory neurons. In each case, the highest classification accuracy of the model achieved 66.7%, 83.3%, and 75%. Figure 6.7 (c) compares the test result when using the different number of excitatory neurons in the output layer. While SNN with 100 excitatory neurons realizes the lowest accuracy, SNN with 400 output neurons reaches the best classification accuracy. As the number of output neurons continues to increase to 1600, the classification

accuracy decreases instead. This result implies that there is no significant positive correlation between the number of output numbers and classification accuracy, which is different from the viewpoint of [1]. Besides, it is easy to find that for 100 and 400 neurons the models have a relatively fast convergence after executing only a few epochs and then improves slowly, while the model with 1600 excitatory neurons gradually converges with more iterations of the training set.

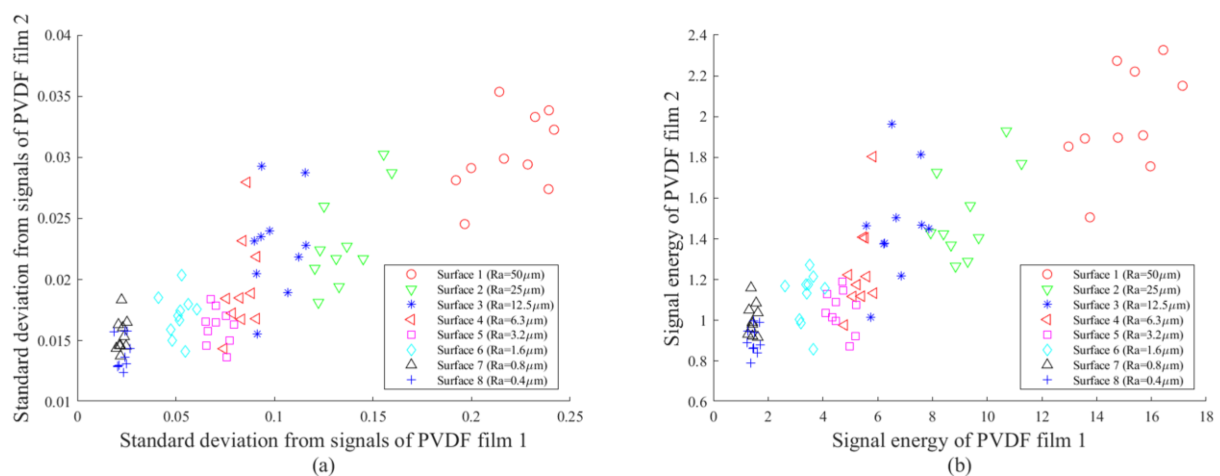


Figure 6.8 (a) Standard deviation features from the tactile signals of the two PVDF films when sliding on eight surfaces with different surface roughness values. (b) Sum of absolute values from the tactile signals of the two PVDF films when sliding on eight surfaces with different surface roughness values.

We made a statistic on the classification results of epoch 11-15, because the test accuracy over those phases performs a stable classification accuracy. Table 6.1 represents the confusion matrix of the classification result. What can be clearly seen in this table is that surface 1 (Ra = 50 μm), surface 2 (Ra = 25 μm), surface 3 (Ra = 12.5 μm) and surface 6 (Ra = 1.6 μm) can be identified accurately by our SNN model. Most misclassifications occur in surface 4 (Ra = 6.3 μm) and surface 7 (Ra = 0.8 μm). Almost all surface 4 were recognized as surface 5 (Ra = 3.2 μm), and the same confusion appeared between surface 7 and surface 8 (Ra = 0.4 μm). In order to observe the sampled tactile signal dataset in more detail, we have drawn the standard

deviation distribution and the signal energy distribution from sampled tactile signals of two PVDF films in Figure 6.8(a) and (b). Equations (6.13) and (6.14) give their definitions. Both figures demonstrate that for tactile signals of PVDF1, surfaces 1, 2, 3, and 6 distribute their range with less overlap with others, while there are overlapping parts that are difficult to separate between surface 4 and surface 5 as well as between surface 6 and surface 7. Compared with tactile signals of PVDF1, tactile signals of PVDF2 have more serious overlap among different surfaces in Figure 6.8(a) and (b) so that it is difficult to separate a single class from a group of data. Thus, even though both signals of PVDF1 and PVDF2 are fused to predict the labels of samples, this method cannot play an effective role, which is also consistent with the result of Figure 6.7 (d). The results indicate that our tactile sensor has limited discriminative ability to effectively pick up the subtle differences of some specific surfaces. In the paper [184], features extracted from discrete wavelet transform in both datasets contribute to the test performance. This way may be explored through SNN with a new encoding method to do frequency domain analysis.

$$S(x(k)) = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}(x(k)-\mu)^2} \tag{6.13}$$

$$E(x(k)) = \sum_{k=1}^{N} x^2(k) \tag{6.14}$$

A comparison of various methods used for surface roughness discrimination is shown in Table 6.2. Compared with the previous works on surface roughness classification [174, 184], this work is different in a few important aspects. First, the adopted SNN does not need the feature selection and extraction stage. While in the kNN or SVM, features need to be obtained first

Table 6.1 Classifier: SNN with 400 output neurons

	S1	S2	S3	S4	S5	S6	S7	S8	Accuracy
S1	15	0	0	0	0	0	0	0	100%
S2	0	15	0	0	0	0	0	0	100%
S3	0	0	15	0	0	0	0	0	100%
S4	0	0	0	1	14	0	0	0	6.7%
S5	0	0	0	1	10	4	0	0	66.7%
S6	0	0	0	0	0	15	0	0	100%
S7	0	0	0	0	0	0	11	4	73.3%
S8	1	0	0	0	0	0	6	8	53.3%
Overall Accuracy: 75%									

S1 = Surface 1 (Ra = 50 μ m) S2 = Surface 2 (Ra = 25 μ m) S3 = Surface 3 (Ra = 12.5 μ m) S4 = Surface 4 (Ra = 6.3 μ m)
S5 = Surface 5 (Ra = 3.2 μ m) S6 = Surface 6 (Ra = 1.6 μ m) S7 = Surface 7 (Ra = 0.8 μ m) S8 = Surface 8 (Ra = 0.4 μ m)

[174, 184], SNN only requires converting the recorded waveform into gray images so as to generate Poisson spike trains as the input for classification, which is simpler and more efficient. Second, the SNN model inspired by the biological neuron structure and parameters used in the SNN model are all within bio-plausible ranges. Finally, the implementation of SNN on hardware is considered to have huge potential: 1. SNN is more hardware friendly than currently popular ANN because SNN uses more energy-efficient "accumulator" units rather than the energy-consuming "multiply-accumulator" [197]; 2. neurons in SNN were triggered only by spiking events. When there is no spike emission, SNN will be silent, which is energy efficient. SNNs exhibit the natural ability of spatiotemporal coding of input, and thus hold the potential advantage of efficient coding through sparse activities, particularly for continuous spatiotemporal inputs; 3. SNN with STDP learning rule is appropriate for online, on-chip learning.

Table 6.2 The highest classification accuracy of different methods

Feature	Classifier	Sensor	Accuracy
SD [174]	kNN (k = 9)	PVDF film 1	82.6%
SF [174]	SVM (RBF)	PVDF film 1	71.2%
SRa [174]	SVM (RBF)	PVDF film 1 & 2	78.8%

PSM [174]	kNN (k = 9)	PVDF film 1	82.5%
SD + PSM [174]	kNN (k = 9)	PVDF film 1	82.5%
SF + PSM [174]	SVM (RBF)	PVDF film 1	72.7%
SRa + PSM [174]	kNN (k = 5)	PVDF film 1	77.8%
SD + PSM [174]	kNN (k = 7)	PVDF film 2	57.9%
Discrete wavelet transform [184]	ELM	PVDF film 1 & 2	97.9%
\	SNN (this paper)	PVDF film 1	83.3%

Note: Statistical features (SF); Signal roughness parameter Ra (SRa); Power spectral magnitudes (PSM); Extreme learning machine (ELM)

6.4 Conclusion

In this chapter, we have explored the possibility of performing tactile surface roughness discrimination with the biologically inspired SNN model. SNNs are very good at handling this time-space information because spiking neurons have a natural internal dynamic system that does not require back connections to handle spatio-temporal signals. Furthermore, due to the low power consumption and low latency of SNN, we can easily integrate the sensor and SNN into the embedded system to build a fast tactile classification system. The Ra value range of sampled surfaces is from 0.4 μm to 50 μm. Ultimately, the test result shows this method can reach the highest 83.3% accuracy by the PVDF1 dataset and 79.3% by both PVDF datasets. This method is robust and suitable for real-time surface roughness discrimination. The insights gained from this study may be of assistance to developing advanced neurorobotics combined with SNN. Further work needs to be done to explore whether SNN can better identify the signal in the frequency domain and by a new encoding method.

6.5 Supplementary

Table 6.3 lists the values of the SNN's parameters.

Table 6.3 Parameters in SNN

Parameter	Description	Value
τ	time constant of LIF neuron	100 ms
τ_{g_e}	time constant of excitatory synapse	1ms
τ_{g_i}	time constant of inhibitory synapse	1ms
u_{rest}	resting membrane potential in excitatory (inhibitory) layer	-65(-60) mV
u_{reset}	reset membrane potential in excitatory (inhibitory) layer	-65(-45) mV
u_e	equilibrium potential of excitatory synapse	0
u_i	equilibrium potentials of inhibitory synapse	-100 mV
A_+	learning rate ($\Delta t \geq 0$)	0.0001
A_-	learning rate ($\Delta t < 0$)	0.01
T	simulation time	350 ms
T_r	refractory period in excitatory (inhibitory) layer	5(2) ms
t	resting time	150 ms
τ_+	time constant ($\Delta t \geq 0$)	20 ms
τ_-	time constant ($\Delta t < 0$)	20 ms
τ_θ	time constant of threshold	1E07
β	scaling constant	1
V_{th}	threshold of excitatory (inhibitory) layer	-52(-40) mV

7 Conclusions

This thesis explores algorithms for brain-inspired SNNs and their applications. Compared to ANNs, SNNs are bio-interpretable, low-power, and low-latency. This is the basis for our focus on the study of SNN algorithms and their applications.

Chapter 1 details the basics of SNNs and the current challenges, while we also summarize the forms of noise present in the brain and its role, and provide an introduction to the current applications of noise in artificial intelligence.

It is still challenging to develop efficient and high-performing learning algorithms for SNNs. In Chapter 2, we propose a novel spiking neuron model (KLIF) to improve the learning ability of SNNs. The neuron model itself can update the slope and width of the surrogate gradient curve during training and selectively delivers membrane potential to spike firing and resetting, which is considered to be more biologically significant. We evaluated our model on both static and neuromorphic datasets. Experiments indicate that KLIF performs much better than current leaky integrate-and-fire (LIF) model, which is most frequently used in SNN, and achieves state-of-the-art performance on those datasets without introducing additional computational cost. Also KLIF increases the firing frequency of individual spiking neuron. The good performance of KLIF can make it completely replace the role of LIF for various tasks.

In Chapter 3, we do a study for the robustness of impulsive neural networks. As more attention is paid to SNNs, security issues become increasingly important. However, there is still a lack of defense methods specifically designed for SNNs. Inspired by neural membrane oscillation, we propose a new bio-plausible neural model that emulates the subthreshold oscillation to enhance the security and robustness of SNNs. Our experiments show that SNNs with neural oscillation models have better resistance to adversarial attacks than ordinary SNNs on kinds of network architectures. Furthermore, we demonstrate the surrogate gradient can affect the effectiveness of adversarial attacks and propose a defense method based on neural oscillation by masking the original surrogate gradients to defend against different types of attacks. The results show that our defense method is comparable to those advanced adversarial training methods used on ANN but requires much less computational costs. To the best of our knowledge, this is the first work that establishes adversarial defense through modifying surrogate gradients on SNNs. As neural oscillations are essential to many neural activities in the biological nervous system, SNN integrated with oscillation mechanism is more bio-plausible than conventional ANN. These findings contribute to our understanding of the relationship between SNN and the biological neural system and provide a basis for optimizing and developing SNN.

Spike-based neuromorphic hardware promises to reduce the energy consumption of image classification and other deep-learning applications, particularly on mobile phones and other edge devices. However, direct training of deep spiking neural networks spends lots of time, and previous methods for converting trained ANNs to spiking neurons were inefficient because the neurons had to emit too many spikes. In Chapter 4, we propose a novel noise based method for faster and efficient SNN training. Our experiments show that ours can reduce training time

by 65%-75% and achieves an inference speed that is more than 100 times faster compared to the previous two methods. We also argue that the neuron model proposed in the paper makes it more bio-plausible.

Sparse representation has attracted great attention because it can greatly save storage resources and find representative features of data in a low-dimensional space. As a result, it may be widely applied in engineering domains including feature extraction, compressed sensing, signal denoising, picture clustering, and dictionary learning, just to name a few. In Chapter 5, we propose a spiking sampling network. This network is composed of spiking neurons and it can dynamically decide which pixel points should be retained and which ones needs to be masked according to the input. Our experiments demonstrate that this approach enables better sparse representation of the original image and facilitates image reconstruction compared to random sampling. We thus use this approach for compressing massive data from the dynamic vision sensor, which greatly reduces the storage requirements for event data.

In the Chapter 6, we apply SNNs to engineering issues in order to process tactile signals. In this study, we show that it is possible to use SNNs to categorise tactile signals obtained from a bionic artificial fingertip that makes contact with a variety of real-world metal surfaces of varying roughnesses. We use a two-layer SNN and train it with an unsupervised learning technique that takes into account STDP. The trained SNN can classify the input tactile signals into metal textures with various levels of surface roughness with an accuracy of over 80%, according to experiments. This establishes the groundwork for using SNNs in robotics, manufacturing, and other engineering domains to process more intricate haptic signals.

In summary, this thesis proposes targeted algorithms for some important challenges faced by

current SNNs such as low training accuracy, slow training speed, and low robustness, while extending their SNNs from being mainly used for classification problems to engineering problems such as sparse coding, and processing of haptic signals. Some innovations in both theory and engineering are made to improve the usability and applications of SNNs.

8 Discussion and Future Work

Future improvements of this thesis can be developed in several areas.

Chapter 3 proposes the defense method against adversarial attacks. These attack samples are generated based on gradients, so the defense is also to make the corresponding interference on the gradients to achieve the defense purpose. Currently, there are also some adversarial attacks that are not based on gradients, and Chapter 3 does not discuss and study for this part, which we will continue to study and research in the future.

In addition, in addition to computer vision, natural language processing (NLP) is also a key area of focus for deep learning, especially like machine translation, text generation, etc. Although we have used SNNs to process temporal spike sequences in Chapter 5, most of the research in this thesis focuses on the processing of image by SNNs. In the future, we will combine SNNs with NLP, given their natural advantages for temporal data.

In addition to the software aspects, the development of SNN-based brain-like chips is also a promising research direction at present. Companies like IBM, and others have specifically developed SNN-based brain-like chips to accelerate networks, and experiments have shown that they have very low power consumption and have great application scenarios in future embedded devices. In the future, we will design and execute our algorithms on hardware to obtain more comprehensive tests.

SNNs still have difficulties in trade-offs between bio-plausibility and performance on large complex datasets. Incorporating working mechanisms inspired by brain mechanisms into SNNs, such as NeuCube [198], which mainly consists of a spike encoder, an unsupervised

module and a supervised module for classification and modelling, could be further combined with our proposed methods in previous chapters to explore the possibility of SNN development. Besides, constructing large models of networks based on spiking neurons is also one of the key directions for the development of SNNs in the future.

While there has been significant progress in developing and understanding SNNs in recent years, there are still many open questions and research directions that can be pursued to further improve their effectiveness and understanding. Some potential research directions and questions for SNNs could be further researched in the future:

- Development of more efficient learning algorithm
- Investigation of the role of different neuron models
- Exploration of different network topologies
- Development of neuromorphic hardware
- Investigation of the role of spatiotemporal patterns
- Application to real-world problems

In conclusion, SNNs are a promising area of research. Developing efficient learning algorithms, optimizing architectures, exploring new applications, developing specialized hardware, and integrating SNNs with other AI techniques are some of the key research directions that can be explored in the future.

References

[1] P.U. Diehl, M. Cook, Unsupervised learning of digit recognition using spike-timing-dependent plasticity, Front Comput Neurosci, 9 (2015) 99.

[2] D. Querlioz, O. Bichler, P. Dollfus, C. Gamrat, Immunity to device variations in a spiking neural network with memristive nanodevices, IEEE Transactions on Nanotechnology, 12 (2013) 288-295.

[3] S.R. Kheradpisheh, M. Ganjtabesh, T. Masquelier, Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition, Neurocomputing, 205 (2016) 382-392.

[4] B. Han, G. Srinivasan, K. Roy, RMP-SNN: Residual Membrane Potential Neuron for Enabling Deeper High-Accuracy and Low-Latency Spiking Neural Network, Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition2020), pp. 13558-13567.

[5] T. Masquelier, S.J. Thorpe, Unsupervised learning of visual features through spike timing dependent plasticity, PLoS Comput Biol, 3 (2007) e31.

[6] G. Srinivasan, P. Panda, K. Roy, Stdp-based unsupervised feature learning using convolution-over-time in spiking neural networks for energy-efficient neuromorphic computing, ACM Journal on Emerging Technologies in Computing Systems (JETC), 14 (2018) 1-12.

[7] K. Roy, A. Jaiswal, P. Panda, Towards spike-based machine intelligence with neuromorphic computing, Nature, 575 (2019) 607-617.

[8] S. Deng, S. Gu, Optimal Conversion of Conventional Artificial Neural Networks to Spiking Neural Networks, arXiv preprint arXiv:2103.00476, (2021).

[9] W. Severa, C.M. Vineyard, R. Dellana, S.J. Verzi, J.B. Aimone, Training deep neural networks for binary communication with the whetstone method, Nature Machine Intelligence, 1 (2019) 86-94.

[10] S. McKennoch, D. Liu, L.G. Bushnell, Fast modifications of the spikeprop algorithm, The 2006 IEEE International Joint Conference on Neural Network Proceedings, (IEEE2006), pp. 3970-3977.

[11] S.M. Bohte, J.N. Kok, J.A. La Poutré, SpikeProp: backpropagation for networks of spiking neurons, ESANN, 48 (2000) 17-37.

[12] F. Zenke, S.M. Bohté, C. Clopath, I.M. Comşa, J. Göltz, W. Maass, T. Masquelier, R. Naud, E.O. Neftci, M.A. Petrovici, Visualizing a joint future of neuroscience and neuromorphic engineering, Neuron, 109 (2021) 571-575.

[13] R. Gutig, H. Sompolinsky, The tempotron: a neuron that learns spike timing-based decisions, Nat Neurosci, 9 (2006) 420-428.

[14] F. Ponulak, A. Kasiński, Supervised learning in spiking neural networks with ReSuMe: sequence learning, classification, and spike shifting, Neural computation, 22 (2010) 467-510.

[15] A. Mohemmed, S. Schliebs, S. Matsuda, N. Kasabov, Span: Spike pattern association neuron for learning spatio-temporal spike patterns, International journal of neural systems, 22 (2012) 1250012.

[16] C. Lee, S.S. Sarwar, P. Panda, G. Srinivasan, K. Roy, Enabling spike-based backpropagation for training deep neural network architectures, Frontiers in neuroscience, 14 (2020).

[17] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, L. Shi, Direct training for spiking neural networks: Faster, larger, better, Proceedings of the AAAI Conference on Artificial Intelligence2019), pp. 1311-1318.

[18] X. Cheng, Y. Hao, J. Xu, B. Xu, LISNN: Improving Spiking Neural Networks with Lateral Interactions for Robust Object Recognition, IJCAI), pp. 1519-1525.

[19] W. Fang, Z. Yu, Y. Chen, T. Masquelier, T. Huang, Y. Tian, Incorporating learnable membrane time constant to enhance learning of spiking neural networks, (July2020).

[20] R. Zimmer, T. Pellegrini, S.F. Singh, T. Masquelier, Technical report: supervised training of convolutional spiking neural networks with PyTorch, arXiv preprint arXiv:1911.10124, (2019).

[21] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, W. Maass, Long short-term memory and learning-to-learn in networks of spiking neurons, arXiv preprint arXiv:1803.09574, (2018).

[22] T. Branco, K. Staras, The probability of neurotransmitter release: variability and feedback control at single synapses, Nature Reviews Neuroscience, 10 (2009) 373-383.

[23] W.H. Calvin, C.F. Stevens, Synaptic noise and other sources of randomness in motoneuron interspike intervals, Journal of neurophysiology, 31 (1968) 574-587.

[24] M. Zhang, H. Qu, X. Xie, J. Kurths, Supervised learning in spiking neural networks with noise-threshold, Neurocomputing, 219 (2017) 333-349.

[25] H.C. Berg, E.M. Purcell, Physics of chemoreception, Biophysical journal, 20 (1977) 193-219.

[26] W. Bialek, S. Setayeshgar, Physical limits to biochemical signaling, Proceedings of the National Academy of Sciences, 102 (2005) 10040-10045.

[27] W. Bialek, Physical limits to sensation and perception, Annual review of biophysics and biophysical chemistry, 16 (1987) 455-478.

[28] R. Azouz, C.M. Gray, Cellular mechanisms contributing to response variability of cortical neurons in vivo, Journal of Neuroscience, 19 (1999) 2209-2223.

[29] M.R. Deweese, A.M. Zador, Shared and private variability in the auditory cortex, Journal of neurophysiology, 92 (2004) 1840-1855.

[30] H. Derksen, A. Verveen, Fluctuations of resting neural membrane potential, Science, 151 (1966) 1388-1389.

[31] A. Verveen, H. Derksen, K. Schick, Voltage fluctuations of neural membrane, Nature, 216 (1967) 588-589.

[32] E. Blair, J. Erlanger, A comparison of the characteristics of axons through their individual electrical responses, American Journal of Physiology-Legacy Content, 106 (1933) 524-564.

[33] P.N. Steinmetz, A. Manwani, C. Koch, M. London, I. Segev, Subthreshold voltage noise due to channel fluctuations in active neuronal membranes, Journal of computational neuroscience, 9 (2000) 133-148.

[34] J.A. White, J.T. Rubinstein, A.R. Kay, Channel noise in neurons, Trends in neurosciences, 23 (2000) 131-137.

[35] M.C. van Rossum, B.J. O'Brien, R.G. Smith, Effects of noise on the spike timing precision of retinal ganglion cells, Journal of neurophysiology, 89 (2003) 2406-2419.

[36] J.T. Rubinstein, Threshold fluctuations in an N sodium channel model of the node of Ranvier, Biophysical journal, 68 (1995) 779-785.

[37] E. Skaugen, L. Walløe, Firing behaviour in a stochastic nerve membrane model based upon the Hodgkin—Huxley equations, Acta Physiologica Scandinavica, 107 (1979) 343-363.

[38] C.C. Chow, J.A. White, Spontaneous action potentials due to channel fluctuations, Biophysical journal, 71 (1996) 3013-3021.

[39] K. Diba, H.A. Lester, C. Koch, Intrinsic noise in cultured hippocampal neurons: experiment and modeling, Journal of Neuroscience, 24 (2004) 9723-9733.

[40] G.A. Jacobson, K. Diba, A. Yaron-Jakoubovitch, Y. Oz, C. Koch, I. Segev, Y. Yarom, Subthreshold voltage noise of rat neocortical pyramidal neurones, The Journal of physiology, 564 (2005) 145-160.

[41] A.D. Dorval, J.A. White, Channel noise is essential for perithreshold oscillations in entorhinal stellate neurons, Journal of neuroscience, 25 (2005) 10025-10028.

[42] M.H. Kole, S. Hallermann, G.J. Stuart, Single Ih channels in pyramidal neuron dendrites: properties, distribution, and impact on action potential output, Journal of Neuroscience, 26 (2006) 1677-1687.

[43] P. Fatt, B. Katz, Spontaneous subthreshold activity at motor nerve endings, The Journal of physiology, 117 (1952) 109.

[44] A. Destexhe, M. Rudolph-Lilith, Neuronal noise (Springer Science & Business Media, 2012).

[45] E.T. Rolls, A. Treves, The neuronal encoding of information in the brain, Prog Neurobiol, 95 (2011) 448-490.

[46] Z.F. Mainen, T.J. Sejnowski, Reliability of spike timing in neocortical neurons, Science, 268 (1995) 1503-1506.

[47] T. Shmiel, R. Drori, O. Shmiel, Y. Ben-Shaul, Z. Nadasdy, M. Shemesh, M. Teicher, M. Abeles, Neurons of the cerebral cortex exhibit precise interspike timing in correspondence to behavior, Proceedings of the National Academy of Sciences, 102 (2005) 18655-18657.

[48] M.A. Montemurro, S. Panzeri, M. Maravall, A. Alenda, M.R. Bale, M. Brambilla, R.S. Petersen, Role of precise spike timing in coding of dynamic vibrissa stimuli in somatosensory thalamus, Journal of neurophysiology, 98 (2007) 1871-1882.

[49] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, science, 220 (1983) 671-680.

[50] A. Krogh, J.A. Hertz, Generalization in a linear perceptron in the presence of noise, Journal of Physics A: Mathematical and General, 25 (1992) 1135.

[51] Y. Shu, A. Hasenstaub, M. Badoual, T. Bal, D.A. McCormick, Barrages of synaptic activity control the gain and sensitivity of cortical neurons, Journal of Neuroscience, 23 (2003) 10388-10401.

[52] R. Benzi, A. Sutera, A. Vulpiani, The mechanism of stochastic resonance, Journal of Physics A: mathematical and general, 14 (1981) L453.

[53] A. Longtin, A. Bulsara, F. Moss, Time-interval sequences in bistable systems and the noise-induced transmission of information by sensory neurons, Physical review letters, 67 (1991) 656.

[54] J.K. Douglass, L. Wilkens, E. Pantazelou, F. Moss, Noise enhancement of information transfer in crayfish mechanoreceptors by stochastic resonance, Nature, 365 (1993) 337-340.

[55] H.A. Braun, H. Wissing, K. Schäfer, M.C. Hirsch, Oscillation and noise determine signal transduction in shark multimodal sensory cells, Nature, 367 (1994) 270-273.

[56] P. Cordo, J.T. Inglis, S. Verschueren, J.J. Collins, D.M. Merfeld, S. Rosenblum, S. Buckley, F. Moss, Noise in human muscle spindles, Nature, 383 (1996) 769-770.

[57] D.F. Russell, L.A. Wilkens, F. Moss, Use of behavioural stochastic resonance by paddle fish for feeding, Nature, 402 (1999) 291-294.

[58] A.A. Priplata, J.B. Niemi, J.D. Harry, L.A. Lipsitz, J.J. Collins, Vibrating insoles and balance control in elderly people, The lancet, 362 (2003) 1123-1124.

[59] J.S. Anderson, I. Lampl, D.C. Gillespie, D. Ferster, The contribution of noise to contrast invariance of orientation tuning in cat visual cortex, Science, 290 (2000) 1968-1972.

[60] S. Kirkpatrick, C.D. Gelatt Jr, M.P. Vecchi, Optimization by simulated annealing, science, 220 (1983) 671-680.

[61] C. Shorten, T.M. Khoshgoftaar, A survey on image data augmentation for deep learning, Journal of big data, 6 (2019) 1-48.

[62] D.P. Kingma, T. Salimans, M. Welling, Variational dropout and the local reparameterization trick, Advances in neural information processing systems, 28 (2015).

[63] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, R. Fergus, Regularization of neural networks using dropconnect, International conference on machine learning, (PMLR2013), pp. 1058-1066.

[64] B. Han, J. Sim, H. Adam, Branchout: Regularization for online ensemble tracking with convolutional neural networks, Proceedings of the IEEE conference on computer vision and pattern recognition2017), pp. 3356-3365.

[65] G. Huang, Y. Sun, Z. Liu, D. Sedra, K.Q. Weinberger, Deep networks with stochastic depth, European conference on computer vision, (Springer2016), pp. 646-661.

[66] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, The journal of machine learning research, 15 (2014) 1929-1958.

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

[68] A. Kurakin, I.J. Goodfellow, S. Bengio, Adversarial examples in the physical world, Artificial intelligence safety and security, (Chapman and Hall/CRC, 2018), pp. 99-112.

[69] S.-M. Moosavi-Dezfooli, A. Fawzi, P. Frossard, Deepfool: a simple and accurate method to fool deep neural networks, Proceedings of the IEEE conference on computer vision and pattern recognition2016), pp. 2574-2582.

[70] X. Liu, M. Cheng, H. Zhang, C.-J. Hsieh, Towards robust neural networks via random self-ensemble, Proceedings of the European Conference on Computer Vision (ECCV)2018), pp. 369-385.

[71] G.W. Ding, Y. Sharma, K.Y.C. Lui, R. Huang, Mma training: Direct input space margin maximization through adversarial training, arXiv preprint arXiv:1812.02637, (2018).

[72] Y. Wang, X. Ma, J. Bailey, J. Yi, B. Zhou, Q. Gu, On the convergence and robustness of adversarial training, arXiv preprint arXiv:2112.08304, (2021).

[73] H. Zhang, Y. Yu, J. Jiao, E. Xing, L. El Ghaoui, M. Jordan, Theoretically principled trade-off between robustness and accuracy, International Conference on Machine Learning, (PMLR2019), pp. 7472-7482.

[74] Y. Wang, D. Zou, J. Yi, J. Bailey, X. Ma, Q. Gu, Improving adversarial robustness requires revisiting misclassified examples, International Conference on Learning Representations2019).

[75] D.P. Kingma, M. Welling, Auto-encoding variational bayes, arXiv preprint arXiv:1312.6114, (2013).

[76] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial networks, Communications of the ACM, 63 (2020) 139-144.

[77] J. Ho, A. Jain, P. Abbeel, Denoising diffusion probabilistic models, Advances in Neural Information Processing Systems, 33 (2020) 6840-6851.

[78] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems, 25 (2012) 1097-1105.

[79] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, Proceedings of the IEEE conference on computer vision and pattern recognition2015), pp. 1-9.

[80] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, Proceedings of the IEEE conference on computer vision and pattern recognition2016), pp. 770-778.

[81] K. He, G. Gkioxari, P. Dollár, R. Girshick, Mask r-cnn, Proceedings of the IEEE international conference on computer vision2017), pp. 2961-2969.

[82] I. Sutskever, O. Vinyals, Q.V. Le, Sequence to sequence learning with neural networks, arXiv preprint arXiv:1409.3215, (2014).

[83] S. Kim, S. Park, B. Na, S. Yoon, Spiking-YOLO: spiking neural network for energy-efficient object detection, Proceedings of the AAAI Conference on Artificial Intelligence2020), pp. 11270-11277.

[84] S. Woźniak, A. Pantazi, T. Bohnstingl, E. Eleftheriou, Deep learning incorporating biologically inspired neural dynamics and in-memory computing, Nature Machine Intelligence, 2 (2020) 325-336.

[85] J. Pei, L. Deng, S. Song, M. Zhao, Y. Zhang, S. Wu, G. Wang, Z. Zou, Z. Wu, W. He, Towards artificial general intelligence with hybrid Tianjic chip architecture, Nature, 572 (2019) 106-111.

[86] Y. Wu, L. Deng, G. Li, J. Zhu, L. Shi, Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks, Front Neurosci, 12 (2018) 331.

[87] J.H. Lee, T. Delbruck, M. Pfeiffer, Training Deep Spiking Neural Networks Using Backpropagation, Front Neurosci, 10 (2016) 508.

[88] E.O. Neftci, H. Mostafa, F. Zenke, Surrogate gradient learning in spiking neural networks, IEEE Signal Processing Magazine, 36 (2019) 61-63.

[89] R. Hecht-Nielsen, Theory of the backpropagation neural network, Neural networks for perception, (Elsevier, 1992), pp. 65-93.

[90] C.J. Schaefer, S. Joshi, Quantizing spiking neural networks with integers, International Conference on Neuromorphic Systems 20202020), pp. 1-8.

[91] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980, (2014).

[92] I. Loshchilov, F. Hutter, Sgdr: Stochastic gradient descent with warm restarts, arXiv preprint arXiv:1608.03983, (2016).

[93] E. Hunsberger, C. Eliasmith, Spiking deep networks with LIF neurons, arXiv preprint arXiv:1510.08829, (2015).

[94] L. Paulun, A. Wendt, N. Kasabov, A retinotopic spiking neural network system for accurate recognition of moving objects using neucube and dynamic vision sensors, Frontiers in Computational Neuroscience, 12 (2018) 42.

[95] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, S.-C. Liu, Conversion of continuous-valued deep networks to efficient event-driven networks for image classification, Frontiers in neuroscience, 11 (2017) 682.

[96] C. Stöckl, W. Maass, Optimized spiking neurons can classify images with high accuracy through temporal coding with two spikes, Nature Machine Intelligence, 3 (2021) 230-238.

[97] W. Zhang, P. Li, Spike-train level backpropagation for training deep recurrent spiking neural networks, arXiv preprint arXiv:1908.06378, (2019).

[98] S.B. Shrestha, G. Orchard, Slayer: Spike layer error reassignment in time, arXiv preprint arXiv:1810.08646, (2018).

[99] J. Kaiser, H. Mostafa, E. Neftci, Synaptic plasticity dynamics for deep continuous local learning (DECOLLE), Frontiers in Neuroscience, 14 (2020) 424.

[100] X. Cheng, Y. Hao, J. Xu, B. Xu, LISNN: Improving Spiking Neural Networks with Lateral Interactions for Robust Object Recognition, IJCAI2020), pp. 1519-1525.

[101] W. He, Y. Wu, L. Deng, G. Li, H. Wang, Y. Tian, W. Ding, W. Wang, Y. Xie, Comparing SNNs and RNNs on neuromorphic vision datasets: Similarities and differences, Neural Networks, 132 (2020) 108-120.

[102] Y. Xing, G. Di Caterina, J. Soraghan, A new spiking convolutional recurrent neural network (SCRNN) with applications to event-based hand gesture recognition, Frontiers in Neuroscience, 14 (2020).

[103] A. Tavanaei, M. Ghodrati, S.R. Kheradpisheh, T. Masquelier, A. Maida, Deep learning in spiking neural networks, Neural networks, 111 (2019) 47-63.

[104] H. Mostafa, B.U. Pedroni, S. Sheik, G. Cauwenberghs, Fast classification using sparsely active spiking networks, 2017 IEEE International Symposium on Circuits and Systems (ISCAS), (IEEE2017), pp. 1-4.

[105] J. Wu, Y. Chua, M. Zhang, Q. Yang, G. Li, H. Li, Deep spiking neural network with spike count based learning rule, 2019 International Joint Conference on Neural Networks (IJCNN), (IEEE2019), pp. 1-6.

[106] L. Liang, X. Hu, L. Deng, Y. Wu, G. Li, Y. Ding, P. Li, Y. Xie, Exploring adversarial attack in spiking neural networks with spike-compatible gradient, IEEE Transactions on Neural Networks and Learning Systems, (2021).

[107] L. Deng, Y. Wu, X. Hu, L. Liang, Y. Ding, G. Li, G. Zhao, P. Li, Y. Xie, Rethinking the performance comparison between SNNS and ANNS, Neural Networks, 121 (2020) 294-307.

[108] W. Maass, Noise as a resource for computation and learning in networks of spiking neurons, Proceedings of the IEEE, 102 (2014) 860-880.

[109] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, L. Shi, Direct training for spiking neural networks: Faster, larger, better, Proceedings of the AAAI Conference on Artificial Intelligence2019), pp. 1311-1318.

[110] G. Haessig, A. Cassidy, R. Alvarez, R. Benosman, G. Orchard, Spiking optical flow for event-based sensors using ibm's truenorth neurosynaptic system, IEEE transactions on biomedical circuits and systems, 12 (2018) 860-870.

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

[112] 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, A million spiking-neuron integrated circuit with a scalable communication network and interface, Science, 345 (2014) 668-673.

[113] A. Kurakin, I. Goodfellow, S. Bengio, Adversarial examples in the physical world, 2016).

[114] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, A. Vladu, Towards deep learning models resistant to adversarial attacks, arXiv preprint arXiv:1706.06083, (2017).

[115] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, P. McDaniel, Ensemble adversarial training: Attacks and defenses, arXiv preprint arXiv:1705.07204, (2017).

[116] W. Xu, D. Evans, Y. Qi, Feature squeezing: Detecting adversarial examples in deep neural networks, arXiv preprint arXiv:1704.01155, (2017).

[117] A. Marchisio, G. Nanfa, F. Khalid, M.A. Hanif, M. Martina, M. Shafique, Is spiking secure? a comparative study on the security vulnerabilities of spiking and deep neural networks, 2020 International Joint Conference on Neural Networks (IJCNN), (IEEE2020), pp. 1-8.

[118] A. Bagheri, O. Simeone, B. Rajendran, Training probabilistic spiking neural networks with first-to-spike decoding, 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), (IEEE2018), pp. 2986-2990.

[119] S. Sharmin, P. Panda, S.S. Sarwar, C. Lee, W. Ponghiran, K. Roy, A comprehensive analysis on adversarial robustness of spiking neural networks, 2019 International Joint Conference on Neural Networks (IJCNN), (IEEE2019), pp. 1-8.

[120] S. Sharmin, N. Rathi, P. Panda, K. Roy, Inherent adversarial robustness of deep spiking neural networks: Effects of discrete input encoding and non-linear activations, European Conference on Computer Vision, (Springer2020), pp. 399-414.

[121] R. El-Allami, A. Marchisio, M. Shafique, I. Alouani, Securing deep spiking neural networks against adversarial attacks through inherent structural parameters, 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), (IEEE2021), pp. 774-779.

[122] A. Krizhevsky, G. Hinton, Learning multiple layers of features from tiny images, (2009).

[123] E. Başar, Brain oscillations in neuropsychiatric disease, Dialogues in clinical neuroscience, 15 (2013) 291.

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

[125] P. Rathore, A. Basak, S.H. Nistala, V. Runkana, Untargeted, Targeted and Universal Adversarial Attacks and Defenses on Time Series, 2020 International Joint Conference on Neural Networks (IJCNN), (IEEE2020), pp. 1-8.

[126] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, J. Li, Boosting adversarial attacks with momentum, Proceedings of the IEEE conference on computer vision and pattern recognition2018), pp. 9185-9193.

[127] Y. Wang, X. Ma, J. Bailey, J. Yi, B. Zhou, Q. Gu, On the Convergence and Robustness of Adversarial Training, ICML2019), pp. 2.

[128] S.M. Bohte, J.N. Kok, H. La Poutre, Error-backpropagation in temporally encoded networks of spiking neurons, Neurocomputing, 48 (2002) 17-37.

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

[130] Y. Cao, Y. Chen, D. Khosla, Spiking deep convolutional neural networks for energy-efficient object recognition, International Journal of Computer Vision, 113 (2015) 54-66.

[131] N. Rathi, 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).

[132] P.U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, M. Pfeiffer, Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing, 2015 International Joint Conference on Neural Networks (IJCNN), (ieee2015), pp. 1-8.

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

[134] D. Purves, R. Cabeza, S.A. Huettel, K.S. LaBar, M.L. Platt, M.G. Woldorff, E.M. Brannon, Cognitive neuroscience (Sunderland: Sinauer Associates, Inc, 2008).

[135] D. Desmaisons, J.-D. Vincent, P.-M. Lledo, Control of action potential timing by intrinsic subthreshold oscillations in olfactory bulb output neurons, Journal of Neuroscience, 19 (1999) 10727-10737.

[136] G. Boehmer, W. Greffrath, E. Martin, S. Hermann, Subthreshold oscillation of the membrane potential in magnocellular neurones of the rat supraoptic nucleus, The Journal of Physiology, 526 (2000) 115.

[137] A.M. Bruckstein, D.L. Donoho, M. Elad, From sparse solutions of systems of equations to sparse modeling of signals and images, SIAM review, 51 (2009) 34-81.

[138] X. Mei, H. Ling, D.W. Jacobs, Sparse representation of cast shadows via ℓ1-regularized least squares, 2009 IEEE 12th International Conference on Computer Vision, (IEEE2009), pp. 583-590.

[139] P. Nagesh, B. Li, A compressive sensing approach for expression-invariant face recognition, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 1518-1525.

[140] B.A. Olshausen, D.J. Field, Sparse coding with an overcomplete basis set: A strategy employed by V1?, Vision research, 37 (1997) 3311-3325.

[141] A. Wagner, J. Wright, A. Ganesh, Z. Zhou, Y. Ma, Towards a practical face recognition system: Robust registration and illumination by sparse representation, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 597-604.

[142] J. Wright, A.Y. Yang, A. Ganesh, S.S. Sastry, Y. Ma, Robust face recognition via sparse representation, IEEE transactions on pattern analysis and machine intelligence, 31 (2008) 210-227.

[143] Z. Zhou, A. Wagner, H. Mobahi, J. Wright, Y. Ma, Face recognition with contiguous occlusion using markov random fields, 2009 IEEE 12th international conference on computer vision, (IEEE2009), pp. 1050-1057.

[144] X. Li, T. Jia, H. Zhang, Expression-insensitive 3D face recognition using sparse representation, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 2575-2582.

[145] J. Yang, J. Wright, T. Huang, Y. Ma, Image super-resolution as sparse representation of raw image patches, 2008 IEEE conference on computer vision and pattern recognition, (IEEE2008), pp. 1-8.

[146] E.E.R. Vidal, Sparse subspace clustering, 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), vol. 002009), pp. 2790-2797.

[147] J.-F. Cai, H. Ji, C. Liu, Z. Shen, Blind motion deblurring from a single image using sparse approximation, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 104-111.

[148] J. Mairal, F. Bach, J. Ponce, G. Sapiro, A. Zisserman, Non-local sparse models for image restoration, 2009 IEEE 12th international conference on computer vision, (IEEE2009), pp. 2272-2279.

[149] J. Mairal, G. Sapiro, M. Elad, Learning multiscale sparse representations for image and video restoration, Multiscale Modeling & Simulation, 7 (2008) 214-241.

[150] V. Cevher, A. Sankaranarayanan, M.F. Duarte, D. Reddy, R.G. Baraniuk, R. Chellappa, Compressive sensing for background subtraction, European Conference on Computer Vision, (Springer2008), pp. 155-168.

[151] M. Dikmen, T.S. Huang, Robust estimation of foreground in surveillance videos by sparse error estimation, 2008 19th International Conference on Pattern Recognition, (IEEE2008), pp. 1-4.

[152] D. Reddy, A. Agrawal, R. Chellappa, Enforcing integrability by error correction using ℓ 1-minimization, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 2350-2357.

[153] C. Wang, S. Yan, L. Zhang, H.-J. Zhang, Multi-label sparse coding for automatic image annotation, 2009 IEEE Conference on Computer Vision and Pattern Recognition, (IEEE2009), pp. 1643-1650.

[154] A.Y. Yang, S. Maji, K. Hong, P. Yan, S.S. Sastry, Distributed compression and fusion of nonnegative sparse signals for multiple-view object recognition, 2009 12th International Conference on Information Fusion, (IEEE2009), pp. 1867-1874.

[155] J. Wright, Y. Ma, J. Mairal, G. Sapiro, T.S. Huang, S. Yan, Sparse representation for computer vision and pattern recognition, Proceedings of the IEEE, 98 (2010) 1031-1044.

[156] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, A.A. Efros, Context encoders: Feature learning by inpainting, Proceedings of the IEEE conference on computer vision and pattern recognition2016), pp. 2536-2544.

[157] M. Chen, A. Radford, R. Child, J. Wu, H. Jun, D. Luan, I. Sutskever, Generative pretraining from pixels, International conference on machine learning, (PMLR2020), pp. 1691-1703.

[158] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, An image is worth 16x16 words: Transformers for image recognition at scale, arXiv preprint arXiv:2010.11929, (2020).

[159] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, R. Girshick, Masked Autoencoders Are Scalable Vision Learners, arXiv preprint arXiv:2111.06377, (2021).

[160] W. Shi, F. Jiang, S. Zhang, D. Zhao, Deep networks for compressed image sensing, 2017 IEEE International Conference on Multimedia and Expo (ICME), (IEEE2017), pp. 877-882.

[161] N. Khan, K. Iqbal, M.G. Martini, Time-Aggregation-Based Lossless Video Encoding for Neuromorphic Vision Sensor Data, IEEE Internet of Things Journal, 8 (2021) 596-609.

[162] I. Schiopu, R.C. Bilcu, Lossless compression of event camera frames, IEEE Signal Processing Letters, 29 (2022) 1779-1783.

[163] N. Khan, K. Iqbal, M.G. Martini, Lossless compression of data from static and mobile dynamic vision sensors-performance and trade-offs, IEEE Access, 8 (2020) 103149-103163.

[164] N. Sengupta, N. Kasabov, Spike-time encoding as a data compression technique for pattern recognition of temporal data, Information Sciences, 406 (2017) 133-145.

[165] N. Khan, K. Iqbal, M.G. Martini, Time-aggregation-based lossless video encoding for neuromorphic vision sensor data, IEEE Internet of Things Journal, 8 (2020) 596-609.

[166] I. Schiopu, R.C. Bilcu, Low-Complexity Lossless Coding of Asynchronous Event Sequences for Low-Power Chip Integration, Sensors, 22 (2022) 10014.

[167] W. Tang, Y. Zhou, H. Zhu, H. Yang, The effect of surface texturing on reducing the friction and wear of steel under lubricated sliding contact, Applied surface science, 273 (2013) 199-204.

[168] W.W. Mayol-Cuevas, J. Juarez-Guerrero, S. Munoz-Gutierrez, A first approach to tactile texture recognition, SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218), (IEEE1998), pp. 4246-4250.

[169] H. Muhammad, C. Recchiuto, C. Oddo, L. Beccai, C. Anthony, M. Adams, M. Carrozza, M. Ward, A capacitive tactile sensor array for surface texture discrimination, Microelectronic Engineering, 88 (2011) 1811-1813.

[170] F. De Boissieu, C. Godin, B. Guilhamat, D. David, C. Serviere, D. Baudois, Tactile texture recognition with a 3-axial force MEMS integrated artificial finger, Robotics: Science and Systems, (Seattle, WA2009), pp. 49-56.

[171] O. Kroemer, C.H. Lampert, J. Peters, Learning dynamic tactile sensing with robust vision-based training, IEEE transactions on robotics, 27 (2011) 545-557.

[172] C.M. Oddo, M. Controzzi, L. Beccai, C. Cipriani, M.C. Carrozza, Roughness encoding for discrimination of surfaces in artificial active-touch, IEEE Transactions on Robotics, 27 (2011) 522-533.

[173] Q. Liang, Z. Yi, Q. Hu, Y. Zhang, Low-Cost Sensor Fusion Technique for Surface Roughness Discrimination With Optical and Piezoelectric Sensors, IEEE Sensors Journal, 17 (2017) 7954-7960.

[174] Z. Yi, Y. Zhang, J. Peters, Bioinspired tactile sensor for surface roughness discrimination, Sensors and Actuators A: Physical, 255 (2017) 46-53.

[175] Y. Zhengkun, Z. Yilei, Recognizing tactile surface roughness with a biomimetic fingertip: A soft neuromorphic approach, Neurocomputing, 244 (2017) 102-111.

[176] C.D. Virgilio G, J.H. Sossa A, J.M. Antelis, L.E. Falcón, Spiking Neural Networks applied to the classification of motor tasks in EEG signals, Neural Networks, 122 (2020) 130-143.

[177] W. Maass, Networks of spiking neurons: the third generation of neural network models, Neural networks, 10 (1997) 1659-1671.

[178] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L.P. Maguire, T.M. McGinnity, A review of learning in biologically plausible spiking neural networks, Neural Netw, 122 (2020) 253-272.

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

[180] Y. Hu, H. Tang, Y. Wang, G. Pan, Spiking deep residual network, arXiv preprint arXiv:1805.01352, (2018).

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

[182] J.M. Antelis, L.E. Falcón, Spiking Neural Networks applied to the classification of motor tasks in EEG signals, Neural Networks, 122 (2020) 130-143.

[183] B. Jaffe, Piezoelectric ceramics (Elsevier, 2012).

[184] L. Qin, Z. Yi, Y. Zhang, Enhanced surface roughness discrimination with optimized features from bio-inspired tactile sensor, Sensors and Actuators A: Physical, 264 (2017) 133-140.

[185] L.N. Cooper, M.F. Bear, The BCM theory of synapse modification at 30: interaction of theory with experiment, Nature Reviews Neuroscience, 13 (2012) 798-810.

[186] K. Pozo, Y. Goda, Unraveling mechanisms of homeostatic synaptic plasticity, Neuron, 66 (2010) 337-351.

[187] Q.-Q. Sun, Experience-dependent intrinsic plasticity in interneurons of barrel cortex layer IV, Journal of neurophysiology, 102 (2009) 2955-2973.

[188] L.C. Yeung, H.Z. Shouval, B.S. Blais, L.N. Cooper, Synaptic homeostasis and input selectivity follow from a calcium-dependent plasticity model, Proceedings of the National Academy of Sciences, 101 (2004) 14943-14948.

[189] W. Zhang, D.J. Linden, The other side of the engram: experience-driven changes in neuronal intrinsic excitability, Nature Reviews Neuroscience, 4 (2003) 885-900.

[190] D. Hebb, Organization of behavior. New York: Wiley, J. Clin. Psychol, 6 (1949) 335-307.

[191] J.-C. Zhang, P.-M. Lau, G.-Q. Bi, Gain in sensitivity and loss in temporal contrast of STDP by dopaminergic modulation at hippocampal synapses, Proceedings of the National Academy of Sciences, 106 (2009) 13028-13033.

[192] Z. Brzosko, W. Schultz, O. Paulsen, Retroactive modulation of spike timing-dependent plasticity by dopamine, Elife, 4 (2015) e09685.

[193] Y. Hao, X. Huang, M. Dong, B. Xu, A biologically plausible supervised learning method for spiking neural networks using the symmetric STDP rule, Neural Netw, 121 (2020) 387-395.

[194] E.T. Rolls, A. Treves, The neuronal encoding of information in the brain, Progress in neurobiology, 95 (2011) 448-490.

[195] B.R. Isett, S.H. Feasel, M.A. Lane, D.E. Feldman, Slip-Based Coding of Local Shape and Texture in Mouse S1, Neuron, 97 (2018) 418-433 e415.

[196] D. Heeger, Poisson model of spike generation, Handout, University of Standford, 5 (2000) 76.

[197] M. Mozafari, S.R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, M. Ganjtabesh, First-spike-based visual categorization using reward-modulated STDP, IEEE transactions on neural networks and learning systems, 29 (2018) 6178-6190.

[198] N.K. Kasabov, NeuCube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data, Neural Networks, 52 (2014) 62-76.

APPENDIX A

This part provides the details of code of SNN models and KLIF neural model discussed in Chapter 2:

```
1.  import torch
2.  import torch.nn as nn
3.  import torch.nn.functional as F
4.  from spikingjelly.clock_driven import functional, layer, surrogate
5.  import math
6.  from torchvision import transforms
7.  import numpy as np
8.  import matplotlib.pyplot as plt
9.  import random
10. from spikingjelly.datasets import play_frame
11.
12.
13.
14. device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
15. thresh = 1.0 # neuronal threshold
16. lens = 0.5 # hyper-parameters of approximate function
17. decay = 0.5 # decay constants
18. num_classes = 10
19. learning_rate = 1e-4
20. alpha = 2.0
21. num_epochs = 100 # max epoch
22.
23.
24. # define approximate firing function
25. class ActFun(torch.autograd.Function):
26.     @staticmethod
27.     def forward(ctx, input):
28.         ctx.save_for_backward(input)
29.         return input.gt(thresh).float()
30.
31.     @staticmethod
32.     def backward(ctx, grad_output):
33.         input, = ctx.saved_tensors
34.         grad_input = grad_output.clone()
35.         temp = alpha / 2 / (1 + (math.pi / 2 * alpha * (input-thresh)).pow_(2))
36.
37.         return grad_input * temp.float()
38.
39.
40. act_fun = ActFun.apply
41.
42.
43. class LIFNode(nn.Module):
44.     def __init__(self):
45.         super(LIFNode, self).__init__()
46.         self.w = torch.nn.Parameter(torch.ones(1), requires_grad=True)
47.         self.func = torch.nn.ReLU()
48.
49.     def forward(self, x, mem, spike,w):
50.         mem = mem * (1. - spike) - mem * decay * (1. - spike) + decay*x
51.         spike = act_fun(mem) # act_fun : approximation firing function
52.
53.         return mem, spike
54.
55.
56. class CIFAR10(nn.Module):
57.     def __init__(self):
58.         super(CIFAR10, self).__init__()
```

```
59.         self.conv1 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,bias=Fa
    lse)
60.         self.conv2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,bias=Fa
    lse)
61.         self.conv3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,bias=Fa
    lse)
62.         self.conv4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,bias=Fa
    lse)
63.         self.conv5 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,bias=Fa
    lse)
64.
65.         self.fc1 = nn.Linear(256*8*8, 2048,bias=False)
66.         self.fc2 = nn.Linear(2048, 100,bias=False)
67.
68.         self.mem0 = LIFNode()
69.         self.mem1 = LIFNode()
70.         self.mem2 = LIFNode()
71.         self.mem3 = LIFNode()
72.         self.mem4 = LIFNode()
73.         self.mem5 = LIFNode()
74.         self.mem6 = LIFNode()
75.         self.mem7 = LIFNode()
76.
77.         self.batch1 = nn.BatchNorm2d(256)
78.         self.batch2 = nn.BatchNorm2d(256)
79.         self.batch3 = nn.BatchNorm2d(256)
80.         self.batch4 = nn.BatchNorm2d(256)
81.         self.batch5 = nn.BatchNorm2d(256)
82.         self.drop1 = layer.Dropout(0.5)
83.         self.drop2 = layer.Dropout(0.5)
84.
85.         self.static_conv = nn.Sequential(
86.             nn.Conv2d(3, 256, kernel_size=3, padding=1, bias=False),
87.             nn.BatchNorm2d(256))
88.         self.static_conv2 = nn.Sequential(
89.             nn.Conv2d(3, 256, kernel_size=3, padding=1, bias=False),
90.             nn.BatchNorm2d(256))
91.         self.static_conv3 = nn.Sequential(
92.             nn.Conv2d(3, 256, kernel_size=3, padding=1, bias=False),
93.             nn.BatchNorm2d(256))
94.
95.
96.
97.     def forward(self, input, batch_size, time_window,train=True):
98.         c0_mem = c0_spike = c0_sumspike = torch.zeros(batch_size,256, 32, 32, devic
    e=device)
99.         c1_mem = c1_spike = c1_sumspike = torch.zeros(batch_size,256, 32, 32, devic
    e=device)
100.                c2_mem = c2_spike = c2_sumspike = torch.zeros(batch_size,256, 32, 32
    , device=device)
101.                c3_mem = c3_spike = c3_sumspike = torch.zeros(batch_size,256, 16, 16
    , device=device)
102.                c4_mem = c4_spike = c4_sumspike = torch.zeros(batch_size,256, 16, 16
    , device=device)
103.                c5_mem = c5_spike = c5_sumspike = torch.zeros(batch_size,256, 16, 16
    , device=device)
104.
105.                h1_mem = h1_spike = h1_sumspike = torch.zeros(batch_size, 2048, devi
    ce=device)
106.                h2_mem = h2_spike = h2_sumspike = torch.zeros(batch_size, 100, devic
    e=device)
107.                h3_mem = h3_spike = h3_sumspike = torch.zeros(batch_size, 10, device
    =device)
108.
109.                input1 = self.static_conv(input)
110.                input2 = self.static_conv2(input)
```

```
111.                input3 = self.static_conv3(input)
112.
113.                input = input1+input2+input3
114.
115.                spike_count = []
116.
117.                for step in range(time_window): # simulation time steps
118.                    c0_mem, c0_spike = self.mem0(input, c0_mem, c0_spike)
119.
120.                    x = self.conv1(c0_spike)
121.                    x = self.batch1(x)
122.                    c1_mem, c1_spike = self.mem1(x, c1_mem, c1_spike)
123.                    x = self.conv2(c1_spike)
124.                    x = self.batch2(x)
125.                    c2_mem, c2_spike = self.mem2(x, c2_mem, c2_spike)

126.                    x = F.max_pool2d(c2_spike, 2)
127.
128.                    x = self.conv3(x)
129.                    x = self.batch3(x)
130.                    c3_mem, c3_spike = self.mem3(x, c3_mem, c3_spike)
131.                    x = self.conv4(c3_spike)
132.                    x = self.batch4(x)
133.                    c4_mem, c4_spike = self.mem4(x, c4_mem, c4_spike)
134.                    x = self.conv5(c4_spike)
135.                    x = self.batch5(x)
136.                    c5_mem, c5_spike = self.mem5(x, c5_mem, c5_spike)

137.                    x = F.max_pool2d(c5_spike, 2)
138.
139.                    x = x.view(batch_size, -1)
140.
141.                    x = self.drop1(x)
142.                    x = self.fc1(x)
143.                    h1_mem, h1_spike = self.mem6(x, h1_mem, h1_spike)
144.
145.                    x = self.drop2(h1_spike)
146.                    x = self.fc2(x)
147.                    h2_mem, h2_spike = self.mem7(x, h2_mem,h2_spike)
148.                    x = F.avg_pool1d(h2_spike.unsqueeze(1), 10)
149.                    h3_sumspike += x.squeeze(1)
150.
151.                outputs = h3_sumspike / time_window
152.
153.                return outputs
154.
155.
156.
157.        class MNIST(nn.Module):
158.            def __init__(self):
159.                super(MNIST, self).__init__()
160.                self.conv1 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
161.                self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
162.
163.                self.fc1 = nn.Linear(128*7*7, 2048,bias=False)
164.                self.fc2 = nn.Linear(2048, 100,bias=False)
165.
166.                self.mem0 = LIFNode()
167.                self.mem1 = LIFNode()
168.                self.mem5 = LIFNode()
169.                self.mem6 = LIFNode()
170.
171.                self.batch1 = nn.BatchNorm2d(128)
172.
```

```
173.                self.drop1 = layer.Dropout(0.5)
174.                self.drop2 = layer.Dropout(0.5)
175.
176.                self.static_conv1 = nn.Sequential(
177.                    nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
178.                    nn.BatchNorm2d(128))
179.                self.static_conv2 = nn.Sequential(
180.                    nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
181.                    nn.BatchNorm2d(128))
182.                self.static_conv3 = nn.Sequential(
183.                    nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
184.                    nn.BatchNorm2d(128))
185.
186.
187.                self.w1 = torch.nn.Parameter(torch.randn(128,28,28), requires_grad=T
    rue)
188.                self.w2 = torch.nn.Parameter(torch.randn(128,14,14), requires_grad=T
    rue)
189.                self.w3 = torch.nn.Parameter(torch.randn(2048), requires_grad=True)

190.                self.w4 = torch.nn.Parameter(torch.randn(100), requires_grad=True)
191.
192.
193.
194.            def forward(self, input,batch_size=50, time_window = 8):
195.                c0_mem = c0_spike = c0_sumspike = torch.zeros(batch_size,128, 28, 28
    , device=device)
196.                c1_mem = c1_spike = c1_sumspike = torch.zeros(batch_size,128, 14, 14
    , device=device)
197.
198.
199.                h1_mem = h1_spike = h1_sumspike = torch.zeros(batch_size, 2048, devi
    ce=device)
200.                h2_mem = h2_spike = h2_sumspike = torch.zeros(batch_size, 100, devic
    e=device)
201.                h3_mem = h3_spike = h3_sumspike = torch.zeros(batch_size, 10, device
    =device)
202.
203.                input1 = self.static_conv1(input)
204.                input2 = self.static_conv2(input)
205.                input3 = self.static_conv3(input)
206.
207.                input = input1+input2+input3
208.
209.                for step in range(time_window): # simulation time steps
210.                    c0_mem, c0_spike = self.mem0(input, c0_mem, c0_spike, self.w1)
211.                    x = F.max_pool2d(c0_spike, 2)
212.
213.                    x = self.conv1(x)
214.                    x = self.batch1(x)
215.                    c1_mem, c1_spike = self.mem1(x, c1_mem, c1_spike, self.w2)
216.                    x = F.max_pool2d(c1_spike, 2)
217.
218.                    x = x.view(batch_size, -1)
219.
220.                    x = self.drop1(x)
221.                    x = self.fc1(x)
222.                    h1_mem, h1_spike = self.mem5(x, h1_mem, h1_spike, self.w3)
223.
224.                    x = self.drop2(h1_spike)
225.                    x = self.fc2(x)
226.                    h2_mem, h2_spike = self.mem6(x, h2_mem,h2_spike, self.w4)
227.
228.                    x = F.avg_pool1d(h2_spike.unsqueeze(1), 10)
229.                    h3_sumspike += x.squeeze(1)
230.                outputs = h3_sumspike / time_window
```

```
231.                return outputs
232.
233.
234.        class N_MNIST(nn.Module):
235.            def __init__(self):
236.                super(N_MNIST, self).__init__()
237.                self.conv1 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
238.                self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
239.
240.                self.fc1 = nn.Linear(128*8*8, 2048,bias=False)
241.                self.fc2 = nn.Linear(2048, 100,bias=False)
242.
243.                self.mem0 = LIFNode()
244.                self.mem1 = LIFNode()
245.                self.mem5 = LIFNode()
246.                self.mem6 = LIFNode()
247.
248.                self.batch0 = nn.BatchNorm2d(128)
249.                self.batch1 = nn.BatchNorm2d(128)
250.
251.                self.drop1 = layer.Dropout(0.5)
252.                self.drop2 = layer.Dropout(0.5)
253.
254.                self.static_conv1 = nn.Sequential(
255.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
256.                    nn.BatchNorm2d(128))
257.                self.static_conv2 = nn.Sequential(
258.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
259.                    nn.BatchNorm2d(128))
260.                self.static_conv3 = nn.Sequential(
261.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
262.                    nn.BatchNorm2d(128))
263.
264.
265.            def forward(self, input,batch_size=50, time_window = 8):
266.                c0_mem = c0_spike = c0_sumspike = torch.zeros(batch_size,128, 34, 34
    , device=device)
267.                c1_mem = c1_spike = c1_sumspike = torch.zeros(batch_size,128, 17, 17
    , device=device)
268.
269.                c3_mem = c3_spike = c3_sumspike = torch.zeros(batch_size,1280, devic
    e=device)
270.                c4_mem = c4_spike = c4_sumspike = torch.zeros(batch_size,128, device
    =device)
271.
272.
273.                h1_mem = h1_spike = h1_sumspike = torch.zeros(batch_size, 2048, devi
    ce=device)
274.                h2_mem = h2_spike = h2_sumspike = torch.zeros(batch_size, 100, devic
    e=device)
275.                h3_mem = h3_spike = h3_sumspike = torch.zeros(batch_size, 10, device
    =device)
276.
277.                input1 = input.permute(1, 0, 2, 3, 4)
278.
279.                for step in range(time_window): # simulation time steps
280.                    x = input1[step]
281.                    x1 = self.static_conv1(x)
282.                    x2 = self.static_conv2(x)
283.                    x3 = self.static_conv3(x)
284.                    x = x1+x2+x3
285.                    c0_mem, c0_spike = self.mem0(x, c0_mem, c0_spike)
286.                    x = F.max_pool2d(c0_spike, 2)
287.
```

```
288.                    x = self.conv1(x)
289.                    x = self.batch1(x)
290.                    c1_mem, c1_spike = self.mem1(x, c1_mem, c1_spike)
291.                    x = F.max_pool2d(c1_spike, 2)
292.
293.                    x = x.view(batch_size, -1)
294.
295.                    x = self.drop1(x)
296.                    x = self.fc1(x)
297.                    h1_mem, h1_spike = self.mem5(x, h1_mem, h1_spike)
298.
299.                    x = self.drop2(h1_spike)
300.                    x = self.fc2(x)
301.                    h2_mem, h2_spike = self.mem6(x, h2_mem,h2_spike)
302.
303.                    x = F.avg_pool1d(h2_spike.unsqueeze(1), 10)
304.                    h3_sumspike += x.squeeze(1)
305.                outputs = h3_sumspike / time_window
306.                return outputs
307.
308.
309.
310.        class CIFAR10_DVS(nn.Module):
311.            def __init__(self):
312.                super(CIFAR10_DVS, self).__init__()
313.                self.conv0 = nn.Conv2d(2, 128, kernel_size=3, stride=1, padding=1,bi
    as=False)
314.                self.conv1 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
315.                self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
316.                self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
317.
318.                self.fc1 = nn.Linear(128*8*8, 512,bias=False)
319.                self.fc2 = nn.Linear(512, 100,bias=False)
320.
321.                self.mem0 = LIFNode()
322.                self.mem1 = LIFNode()
323.                self.mem2 = LIFNode()
324.                self.mem3 = LIFNode()
325.                self.mem5 = LIFNode()
326.                self.mem6 = LIFNode()
327.
328.                self.batch0 = nn.BatchNorm2d(128)
329.                self.batch1 = nn.BatchNorm2d(128)
330.                self.batch2 = nn.BatchNorm2d(128)
331.                self.batch3 = nn.BatchNorm2d(128)
332.
333.                self.drop1 = layer.Dropout(0.5)
334.                self.drop2 = layer.Dropout(0.5)
335.                '''
336.                self.static_conv1 = nn.Sequential(
337.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
338.                    nn.BatchNorm2d(128))
339.                self.static_conv2 = nn.Sequential(
340.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
341.                    nn.BatchNorm2d(128))
342.                self.static_conv3 = nn.Sequential(
343.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
344.                    nn.BatchNorm2d(128)) '''
345.
346.            def forward(self, input,batch_size, time_window,train=True):
347.                c0_mem = c0_spike  = torch.zeros(128, 128, 128, device=device)
348.                c1_mem = c1_spike  = torch.zeros(128, 64, 64, device=device)
349.                c2_mem = c2_spike  = torch.zeros(128, 32, 32, device=device)
```

```
350.                c3_mem = c3_spike  = torch.zeros(128, 16, 16, device=device)
351.
352.                h1_mem = h1_spike  = torch.zeros(512, device=device)
353.                h2_mem = h2_spike  = torch.zeros(100, device=device)
354.                h3_mem = h3_spike  = h3_sumspike = torch.zeros(10, device=device)
355.
356.                input1 = input.permute(1, 0, 2, 3, 4)
357.
358.                spike_count = []
359.
360.                for step in range(time_window): # simulation time steps
361.
362.                    x = input1[step]
363.                    x = self.conv0(x)
364.                    x = self.batch0(x)
365.
366.                    c0_mem, c0_spike = self.mem0(x, c0_mem, c0_spike)
367.                    x = F.max_pool2d(c0_spike, 2)
368.
369.                    x = self.conv1(x)
370.                    x = self.batch1(x)
371.                    c1_mem, c1_spike = self.mem1(x, c1_mem, c1_spike)
372.                    x = F.max_pool2d(c1_spike, 2)
373.
374.                    x = self.conv2(x)
375.                    x = self.batch2(x)
376.                    c2_mem, c2_spike = self.mem2(x, c2_mem, c2_spike)
377.                    x = F.max_pool2d(c2_spike, 2)
378.
379.                    x = self.conv3(x)
380.                    x = self.batch3(x)
381.                    c3_mem, c3_spike = self.mem3(x, c3_mem, c3_spike)
382.                    x = F.max_pool2d(c3_spike, 2)
383.
384.                    x = x.view(batch_size, -1)
385.
386.                    x = self.drop1(x)
387.                    x = self.fc1(x)
388.                    h1_mem, h1_spike = self.mem5(x, h1_mem, h1_spike)
389.
390.                    x = self.drop2(h1_spike)
391.                    x = self.fc2(x)
392.                    h2_mem, h2_spike = self.mem6(x, h2_mem,h2_spike)
393.
394.                    x = F.avg_pool1d(h2_spike.unsqueeze(1), 10)
395.                    h3_sumspike = h3_sumspike + x.squeeze(1)
396.                outputs = h3_sumspike / time_window
397.
398.
399.                return outputs
400.
401.        class DVS_GESTURE(nn.Module):
402.            def __init__(self):
403.                super(DVS_GESTURE, self).__init__()
404.                self.conv0 = nn.Conv2d(2, 128, kernel_size=3, stride=1, padding=1,bi
    as=False)
405.                self.conv1 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
406.                self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
407.                self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
408.                self.conv4 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1,
    bias=False)
409.
410.                self.fc1 = nn.Linear(128*4*4, 512,bias=False)
```

```
411.                self.fc2 = nn.Linear(512, 110,bias=False)
412. 
413.                self.mem0 = LIFNode()
414.                self.mem1 = LIFNode()
415.                self.mem2 = LIFNode()
416.                self.mem3 = LIFNode()
417.                self.mem4 = LIFNode()
418.                self.mem5 = LIFNode()
419.                self.mem6 = LIFNode()
420. 
421.                self.batch0 = nn.BatchNorm2d(128)
422.                self.batch1 = nn.BatchNorm2d(128)
423.                self.batch2 = nn.BatchNorm2d(128)
424.                self.batch3 = nn.BatchNorm2d(128)
425.                self.batch4 = nn.BatchNorm2d(128)
426. 
427.                self.drop1 = layer.Dropout(0.5)
428.                self.drop2 = layer.Dropout(0.5)
429. 
430. 
431.                self.static_conv1 = nn.Sequential(
432.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
433.                    nn.BatchNorm2d(128))
434.                self.static_conv2 = nn.Sequential(
435.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
436.                    nn.BatchNorm2d(128))
437.                self.static_conv3 = nn.Sequential(
438.                    nn.Conv2d(2, 128, kernel_size=3, padding=1, bias=False),
439.                    nn.BatchNorm2d(128))
440. 
441. 
442. 
443.            def forward(self, input,batch_size,time_window,train=True):
444.                c0_mem = c0_spike  = torch.zeros(batch_size,128, 128, 128, device=de
    vice)
445.                c1_mem = c1_spike  = torch.zeros(batch_size,128, 64, 64, device=devi
    ce)
446.                c2_mem = c2_spike  = torch.zeros(batch_size,128, 32, 32, device=devi
    ce)
447.                c3_mem = c3_spike  = torch.zeros(batch_size,128, 16, 16, device=devi
    ce)
448.                c4_mem = c4_spike  = torch.zeros(batch_size,128, 8, 8, device=device
    )
449. 
450.                h1_mem = h1_spike  = torch.zeros(batch_size, 512, device=device)
451.                h2_mem = h2_spike  = torch.zeros(batch_size, 110, device=device)
452.                h3_mem = h3_spike  = h3_sumspike = torch.zeros(batch_size, 11, devic
    e=device)
453. 
454.                spike_count = []
455.                input1 = input.permute(1, 0, 2, 3, 4)
456. 
457. 
458.                for step in range(time_window): # simulation time steps
459.                    x = input1[step]
460.                    x1 = self.static_conv1(x)
461.                    x2 = self.static_conv2(x)
462.                    x3 = self.static_conv3(x)
463.                    x = x1+x2+x3
464. 
465.                    c0_mem, c0_spike = self.mem0(x, c0_mem, c0_spike)
466.                    x = F.max_pool2d(c0_spike, 2)
467. 
468.                    x = self.conv1(x)
469.                    x = self.batch1(x)
470.                    c1_mem, c1_spike = self.mem1(x, c1_mem, c1_spike)
```

```
471.                    x = F.max_pool2d(c1_spike, 2)
472.
473.                    x = self.conv2(x)
474.                    x = self.batch2(x)
475.                    c2_mem, c2_spike = self.mem2(x, c2_mem, c2_spike)
476.                    x = F.max_pool2d(c2_spike, 2)
477.
478.                    x = self.conv3(x)
479.                    x = self.batch3(x)
480.                    c3_mem, c3_spike = self.mem3(x, c3_mem, c3_spike)
481.                    x = F.max_pool2d(c3_spike, 2)
482.
483.                    x = self.conv4(x)
484.                    x = self.batch4(x)
485.                    c4_mem, c4_spike = self.mem4(x, c4_mem, c4_spike)
486.                    x = F.max_pool2d(c4_spike, 2)
487.
488.                    x = x.view(batch_size, -1)
489.
490.                    x = self.drop1(x)
491.                    x = self.fc1(x)
492.                    h1_mem, h1_spike = self.mem5(x, h1_mem, h1_spike)
493.
494.                    x = self.drop2(h1_spike)
495.                    x = self.fc2(x)
496.                    h2_mem, h2_spike = self.mem6(x, h2_mem,h2_spike)
497.
498.                    x = F.avg_pool1d(h2_spike.unsqueeze(1), 10)
499.                    h3_sumspike += x.squeeze(1)
500.                outputs = h3_sumspike / time_window
501.
502.
503.                return outputs
```

APPENDIX B

This part provides the details of code of SNN models and neural oscillation model discussed in Chapter 3:

```
1.  import torch
2.  import torch.nn as nn
3.  import torch.nn.functional as F
4.  from spikingjelly.clock_driven import functional, layer, surrogate
5.  import math
6.  from torchvision import transforms
7.  import numpy as np
8.  import matplotlib.pyplot as plt
9.  import random
10. from spikingjelly import visualizing
11.
12.
13. device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
14. thresh = 1.0  # neuronal threshold
15. lens = 0.5 # hyper-parameters of approximate function
16. decay = 0.5 # decay constants
17. learning_rate = 1e-3
18. alpha = 3
19. num_epochs = 100 # max epoch
20. time_window= 8
21. batch_size = 50
22.
23.
24. # define approximate firing function
25. class ActFun(torch.autograd.Function):
26.     @staticmethod
27.     def forward(ctx, input):
28.         ctx.save_for_backward(input)
29.         return input.gt(thresh).float()
30.
31.     @staticmethod
32.     def backward(ctx, grad_output):
33.         input, = ctx.saved_tensors
34.         grad_input = grad_output.clone()
35.         temp = alpha / 2 / (1 + (math.pi / 2 * alpha * (input-thresh)).pow_(2))
36.
37.         return grad_input * temp.float(),None
38.
39.
40.
41. act_fun = ActFun.apply
42.
43.
44. class LIFNode(nn.Module):
45.     def __init__(self):
46.         super(LIFNode, self).__init__()
47.         self.func = torch.nn.LeakyReLU(negative_slope=-0.03)
48.
49.     def forward(self, ops, x, mem, spike):
50.         mem = mem * (1. - spike) - mem * decay * (1. - spike) + ops(x)
51.         mem = self.func(mem+torch.randn(mem.size(),device=device)*0.5)
52.
53.         spike = act_fun(mem) # act_fun : approximation firing function
54.         return mem, spike
55.
56.
57.
58. class VGG_SNN(nn.Module):
```

```
59.     def __init__(self):
60.         super(VGG_SNN, self).__init__()
61.         self.conv1 = nn.Conv2d(in_channels=3,out_channels=64,kernel_size=(3, 3),str
    ide=(1, 1),padding=1,bias=False)
62.         self.conv2 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=(3, 3),st
    ride=(1, 1),padding=1,bias=False)
63.
64.         self.conv3 = nn.Conv2d(in_channels=64,out_channels=128,kernel_size=(3, 3),s
    tride=(1, 1),padding=1,bias=False)
65.         self.conv4 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
66.
67.         self.conv5 = nn.Conv2d(in_channels=128,out_channels=256,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
68.         self.conv6 = nn.Conv2d(in_channels=256,out_channels=256,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
69.         self.conv7 = nn.Conv2d(in_channels=256,out_channels=256,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
70.
71.         self.conv8 = nn.Conv2d(in_channels=256,out_channels=512,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
72.         self.conv9 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size=(3, 3),
    stride=(1, 1),padding=1,bias=False)
73.         self.conv10 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size=(3, 3)
    ,stride=(1, 1),padding=1,bias=False)
74.
75.         self.conv11 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size=(3, 3)
    ,stride=(1, 1),padding=1,bias=False)
76.         self.conv12 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size=(3, 3)
    ,stride=(1, 1),padding=1,bias=False)
77.         self.conv13 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size=(3, 3)
    ,stride=(1, 1),padding=1,bias=False)
78.
79.         self.fc1 = nn.Linear(512, 512,bias=False)
80.         self.fc2 = nn.Linear(512, 512,bias=False)
81.         self.fc3 = nn.Linear(512, 10,bias=False)
82.
83.         self.batch1 = nn.BatchNorm2d(64)
84.         self.batch2 = nn.BatchNorm2d(64)
85.         self.batch3 = nn.BatchNorm2d(128)
86.         self.batch4 = nn.BatchNorm2d(128)
87.         self.batch5 = nn.BatchNorm2d(256)
88.         self.batch6 = nn.BatchNorm2d(256)
89.         self.batch7 = nn.BatchNorm2d(256)
90.         self.batch8 = nn.BatchNorm2d(512)
91.         self.batch9 = nn.BatchNorm2d(512)
92.         self.batch10 = nn.BatchNorm2d(512)
93.         self.batch11 = nn.BatchNorm2d(512)
94.         self.batch12 = nn.BatchNorm2d(512)
95.         self.batch13 = nn.BatchNorm2d(512)
96.         self.drop1 = layer.Dropout(0.5)
97.         self.drop2 = layer.Dropout(0.5)
98.
99.         self.mem = LIFNode()
100.
101.                self.flatten = nn.Flatten()
102.
103.                self.s_list = torch.tensor([], device=device)
104.                self.v_list = torch.tensor([], device=device)
105.
106.
107.            def forward(self, input):
108.
109.
110.                c1_mem = c1_spike = c1_sumspike = c1_minus = torch.zeros(64, 32, 32,
     device=device)
```

```
111.                c2_mem = c2_spike = c2_sumspike = c2_minus = torch.zeros(64, 32, 32,
     device=device)
112.
113.                c3_mem = c3_spike = c3_sumspike = c3_minus = torch.zeros(128, 16, 16
    , device=device)
114.                c4_mem = c4_spike = c4_sumspike = c4_minus = torch.zeros(128, 16, 16
    , device=device)
115.
116.                c5_mem = c5_spike = c5_sumspike = c5_minus = torch.zeros(256, 8, 8,
    device=device)
117.                c6_mem = c6_spike = c6_sumspike = c6_minus = torch.zeros(256, 8, 8,
    device=device)
118.                c7_mem = c7_spike = c7_sumspike = c7_minus = torch.zeros(256, 8, 8,
    device=device)
119.
120.                c8_mem = c8_spike = c8_sumspike = c8_minus = torch.zeros(512, 4, 4,
    device=device)
121.                c9_mem = c9_spike = c9_sumspike = c9_minus = torch.zeros(512, 4, 4,
    device=device)
122.                c10_mem = c10_spike = c10_sumspike = c10_minus = torch.zeros(512, 4,
     4, device=device)
123.
124.                c11_mem = c11_spike = c11_sumspike = c11_minus = torch.zeros(512, 2,
     2, device=device)
125.                c12_mem = c12_spike = c12_sumspike = c12_minus = torch.zeros(512, 2,
     2, device=device)
126.                c13_mem = c13_spike = c13_sumspike = c13_minus = torch.zeros(512, 2,
     2, device=device)
127.
128.
129.                h1_mem = h1_spike = h1_sumspike = h1_minus = torch.zeros(512, device
    =device)
130.                h2_mem = h2_spike = h2_sumspike = h2_minus = torch.zeros(512, device
    =device)
131.                h3_mem = h3_spike = h3_sumspike = h3_minus = torch.zeros(10, device=
    device)
132.
133.
134.                for step in range(time_window): # simulation time steps
135.                    # block1
136.                    c1_mem, c1_spike = self.mem(self.batch1,self.conv1(input), c1_me
    m, c1_spike)
137.                    c2_mem, c2_spike = self.mem(self.batch2,self.conv2(c1_spike), c2
    _mem, c2_spike)
138.                    x = F.avg_pool2d(c2_spike, 2)
139.
140.
141.                    #block2
142.                    c3_mem, c3_spike = self.mem(self.batch3,self.conv3(x.float()), c
    3_mem, c3_spike)
143.                    c3_sumspike = c3_sumspike + c3_spike
144.                    c4_mem, c4_spike = self.mem(self.batch4,self.conv4(c3_spike), c4
    _mem, c4_spike)
145.                    x = F.avg_pool2d(c4_spike, 2)
146.
147.                    #block3
148.                    c5_mem, c5_spike = self.mem(self.batch5,self.conv5(x.float()), c
    5_mem, c5_spike,)
149.                    c6_mem, c6_spike = self.mem(self.batch6,self.conv6(c5_spike), c6
    _mem, c6_spike,)
150.                    c7_mem, c7_spike = self.mem(self.batch7,self.conv7(c6_spike), c7
    _mem, c7_spike)
151.                    x = F.avg_pool2d(c7_spike, 2)
152.
153.
154.                    #block4
```

```
155.                    c8_mem, c8_spike = self.mem(self.batch8,self.conv8(x.float()), c
    8_mem, c8_spike)
156.                    c9_mem, c9_spike = self.mem(self.batch9,self.conv9(c8_spike), c9
    _mem, c9_spike)
157.                    c10_mem, c10_spike = self.mem(self.batch10,self.conv10(c9_spike)
    , c10_mem, c10_spike)
158.                    x = F.avg_pool2d(c10_spike, 2)
159.
160.                    #block5
161.                    c11_mem, c11_spike = self.mem(self.batch11,self.conv11(x.float()
    ), c11_mem, c11_spike)
162.                    c12_mem, c12_spike = self.mem(self.batch12,self.conv12(c11_spike
    ), c12_mem, c12_spike)
163.                    c13_mem, c13_spike = self.mem(self.batch13,self.conv13(c12_spike
    ), c13_mem, c13_spike)
164.                    x = F.avg_pool2d(c13_spike, 2)
165.
166.                    x = self.flatten(x)
167.                    h1_mem, h1_spike = self.mem(self.fc1,self.drop1(x), h1_mem, h1_s
    pike)
168.                    h2_mem, h2_spike = self.mem(self.fc2,self.drop2(h1_spike), h2_me
    m, h2_spike)
169.                    h3_mem, h3_spike = self.mem(self.fc3,h2_spike, h3_mem, h3_spike)

170.                    h3_sumspike = h3_sumspike + h3_spike
171.
172.
173.                outputs = h3_sumspike/time_window
174.
175.                return outputs
176.
177.
178.        class ResidualBlock(nn.Module):
179.            def __init__(self, inchannel, outchannel, stride=2):
180.                super(ResidualBlock, self).__init__()
181.
182.                self.shortcut = nn.Sequential(
183.                    nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride, b
    ias=False),
184.                    nn.BatchNorm2d(outchannel)
185.                )
186.            def forward(self, x):
187.                out = self.shortcut(x)
188.                return out
189.
190.
191.        class ResnetSnn(nn.Module):
192.            def __init__(self):
193.                super(ResnetSnn, self).__init__()
194.                self.conv0 = nn.Conv2d(in_channels=3,out_channels=64,kernel_size=(3,
     3),stride=(1, 1),padding=1,bias=False)
195.                self.conv1 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=(3
    , 3),stride=(1, 1),padding=1,bias=False)
196.                self.conv2 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=(3
    , 3),stride=(1, 1),padding=1,bias=False)
197.                self.conv3 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=(3
    , 3),stride=(1, 1),padding=1,bias=False)
198.                self.conv4 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=(3
    , 3),stride=(1, 1),padding=1,bias=False)
199.
200.                self.conv5 = nn.Conv2d(in_channels=64,out_channels=128,kernel_size=(
    3, 3),stride=(2, 2),padding=1,bias=False)
201.                self.conv6 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=
    (3, 3),stride=(1, 1),padding=1,bias=False)
202.                self.conv7 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=
    (3, 3),stride=(1, 1),padding=1,bias=False)
```

```
203.                self.conv8 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=
    (3, 3),stride=(1, 1),padding=1,bias=False)
204.
205.                self.conv9 = nn.Conv2d(in_channels=128,out_channels=256,kernel_size=
    (3, 3),stride=(2, 2),padding=1,bias=False)
206.                self.conv10 = nn.Conv2d(in_channels=256,out_channels=256,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
207.                self.conv11 = nn.Conv2d(in_channels=256,out_channels=256,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
208.                self.conv12 = nn.Conv2d(in_channels=256,out_channels=256,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
209.
210.                self.conv13 = nn.Conv2d(in_channels=256,out_channels=512,kernel_size
    =(3, 3),stride=(2, 2),padding=1,bias=False)
211.                self.conv14 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
212.                self.conv15 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
213.                self.conv16 = nn.Conv2d(in_channels=512,out_channels=512,kernel_size
    =(3, 3),stride=(1, 1),padding=1,bias=False)
214.
215.
216.                self.fc1 = nn.Linear(512, 10,bias=False)
217.
218.
219.                self.batch0 = nn.BatchNorm2d(64)
220.                self.batch1 = nn.BatchNorm2d(64)
221.                self.batch2 = nn.BatchNorm2d(64)
222.                self.batch3 = nn.BatchNorm2d(64)
223.                self.batch4 = nn.BatchNorm2d(64)
224.                self.batch5 = nn.BatchNorm2d(128)
225.                self.batch6 = nn.BatchNorm2d(128)
226.                self.batch7 = nn.BatchNorm2d(128)
227.                self.batch8 = nn.BatchNorm2d(128)
228.                self.batch9 = nn.BatchNorm2d(256)
229.                self.batch10 = nn.BatchNorm2d(256)
230.                self.batch11 = nn.BatchNorm2d(256)
231.                self.batch12 = nn.BatchNorm2d(256)
232.                self.batch13 = nn.BatchNorm2d(512)
233.                self.batch14 = nn.BatchNorm2d(512)
234.                self.batch15 = nn.BatchNorm2d(512)
235.                self.batch16 = nn.BatchNorm2d(512)
236.                self.batch17 = nn.BatchNorm1d(10)
237.
238.                self.drop1 = layer.Dropout(0.5)
239.                self.drop2 = layer.Dropout(0.5)
240.
241.                self.mem = LIFNode()
242.                self.flatten = nn.Flatten()
243.
244.                self.block1 = ResidualBlock(64,128)
245.                self.block2 = ResidualBlock(128,256)
246.                self.block3 = ResidualBlock(256,512)
247.
248.
249.                self.static_conv = nn.Sequential(
250.                    nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
251.                    nn.BatchNorm2d(64)
252.                    )
253.
254.                self.avg1 = nn.Conv2d(in_channels=3,out_channels=128,kernel_size=(2,
     2),stride=(2, 2),padding=0,bias=False)
255.                self.avg2 = nn.Conv2d(in_channels=3,out_channels=256,kernel_size=(4,
     4),stride=(4, 4),padding=0,bias=False)
256.                self.avg3 = nn.Conv2d(in_channels=3,out_channels=512,kernel_size=(8,
     8),stride=(8, 8),padding=0,bias=False)
```

```
257.
258.
259.            def forward(self, input):
260.                c0_mem = c0_spike = c0_sumspike = c0_minus = torch.zeros(64, 32, 32,
     device=device)
261.
262.                c1_mem = c1_spike = c1_sumspike = c1_minus = torch.zeros(64, 32, 32,
     device=device)
263.                c2_mem = c2_spike = c2_sumspike = c2_minus = torch.zeros(64, 32, 32,
     device=device)
264.                c3_mem = c3_spike = c3_sumspike = c3_minus = torch.zeros(64, 32, 32,
     device=device)
265.                c4_mem = c4_spike = c4_sumspike = c4_minus = torch.zeros(64, 32, 32,
     device=device)
266.
267.                c5_mem = c5_spike = c5_sumspike = c5_minus = torch.zeros(128, 16, 16
    , device=device)
268.                c6_mem = c6_spike = c6_sumspike = c6_minus = torch.zeros(128, 16, 16
    , device=device)
269.                c7_mem = c7_spike = c7_sumspike = c7_minus = torch.zeros(128, 16, 16
    , device=device)
270.                c8_mem = c8_spike = c8_sumspike = c8_minus = torch.zeros(128, 16, 16
    , device=device)
271.
272.                c9_mem = c9_spike = c9_sumspike = c9_minus = torch.zeros(256, 8, 8,
    device=device)
273.                c10_mem = c10_spike = c10_sumspike = c10_minus = torch.zeros(256, 8,
     8, device=device)
274.                c11_mem = c11_spike = c11_sumspike = c11_minus = torch.zeros(256, 8,
     8, device=device)
275.                c12_mem = c12_spike = c12_sumspike = c12_minus = torch.zeros(256, 8,
     8, device=device)
276.
277.                c13_mem = c13_spike = c13_sumspike = c13_minus = torch.zeros(512, 4,
     4, device=device)
278.                c14_mem = c14_spike = c14_sumspike = c14_minus = torch.zeros(512, 4,
     4, device=device)
279.                c15_mem = c15_spike = c15_sumspike = c15_minus = torch.zeros(512, 4,
     4, device=device)
280.                c16_mem = c16_spike = c16_sumspike = c16_minus = torch.zeros(512, 4,
     4, device=device)
281.
282.
283.                h1_mem = h1_spike = h1_sumspike = h1_minus = torch.zeros(10, device=
    device)
284.                h2_mem = h2_spike = h2_sumspike = h2_minus = torch.zeros(10, device=
    device)
285.                h3_mem = h3_spike = h3_sumspike = h3_minus = torch.zeros(10, device=
    device)
286.
287.
288.
289.                for step in range(time_window): # simulation time steps
290.                    c0_mem, c0_spike = self.mem(self.batch0,self.conv0(input), c0_me
    m, c0_spike)
291.
292.                    #block1
293.                    c1_mem, c1_spike = self.mem(self.batch1,self.conv1(c0_spike), c1
    _mem, c1_spike)
294.                    c2_mem, c2_spike = self.mem(self.batch2,self.conv2(c1_spike), c2
    _mem+c0_spike, c2_spike)#+input1
295.                    c3_mem, c3_spike = self.mem(self.batch3,self.conv3(c2_spike), c3
    _mem, c3_spike)
296.                    c4_mem, c4_spike = self.mem(self.batch4,self.conv4(c3_spike), c4
    _mem+c2_spike, c4_spike)
297.
```

```
298.                    #block2
299.                    c5_mem, c5_spike = self.mem(self.batch5,self.conv5(c4_spike), c5
    _mem, c5_spike)
300.                    c6_mem, c6_spike = self.mem(self.batch6,self.conv6(c5_spike), c6
    _mem+self.block1(c4_spike), c6_spike)
301.                    c7_mem, c7_spike = self.mem(self.batch7,self.conv7(c6_spike), c7
    _mem, c7_spike)
302.                    c8_mem, c8_spike = self.mem(self.batch8,self.conv8(c7_spike), c8
    _mem+c6_spike, c8_spike)
303.
304.                    #block3
305.                    c9_mem, c9_spike = self.mem(self.batch9,self.conv9(c8_spike), c9
    _mem, c9_spike)
306.                    c10_mem, c10_spike = self.mem(self.batch10,self.conv10(c9_spike)
    , c10_mem+self.block2(c8_spike), c10_spike)
307.                    c11_mem, c11_spike = self.mem(self.batch11,self.conv11(c10_spike
    ), c11_mem, c11_spike)
308.                    c12_mem, c12_spike = self.mem(self.batch12,self.conv12(c11_spike
    ), c12_mem+c10_spike, c12_spike)
309.
310.                    #block4
311.                    c13_mem, c13_spike = self.mem(self.batch13,self.conv13(c12_spike
    ), c13_mem, c13_spike)
312.                    c14_mem, c14_spike = self.mem(self.batch14,self.conv14(c13_spike
    ), c14_mem+self.block3(c12_spike), c14_spike)
313.                    c15_mem, c15_spike = self.mem(self.batch15,self.conv15(c14_spike
    ), c15_mem, c15_spike)
314.                    c16_mem, c16_spike = self.mem(self.batch16,self.conv16(c15_spike
    ), c16_mem+c14_spike, c16_spike)
315.                    x = c16_spike
316.
317.
318.                    x = F.avg_pool2d(x, 4)
319.                    x = self.flatten(x)
320.
321.                    h1_mem, h1_spike = self.mem(self.batch17,self.fc1(x), h1_mem, h1
    _spike)
322.
323.                    h1_sumspike = h1_sumspike+h1_spike
324.
325.                outputs = h1_sumspike / time_window
326.
327.                return outputs
```

Training alternative neural oscillation model:

```
1.  class LIFNode(nn.Module):
2.      def __init__(self):
3.          super(LIFNode, self).__init__()
4.          self.func = torch.nn.LeakyReLU(negative_slope=-0.03)
5.          self.bias0 = torch.nn.Parameter(torch.zeros(1), requires_grad=True)
6.          self.bias1 = torch.nn.Parameter(torch.zeros(1), requires_grad=True)
7.
8.      def forward(self, ops, x, mem, spike,train=True):
9.          mem = mem * (1. - spike) - mem * decay * (1. - spike) + ops(x)
10.
11.         if train==True:
12.           mem1 = self.func(mem+torch.rand(mem.size(),device=device)-
    torch.ones(1,device=device)*0.2 )
13.           mem2 = self.func(mem+torch.sin(mem+self.bias1)+self.bias0)
14.         else:
```

```
15.           mem2 = self.func(mem+torch.rand(mem.size(),device=device)-
    torch.ones(1,device=device)*0.2 )
16.           mem1 = self.func(mem+torch.sin(mem+self.bias1)+self.bias0)
17.         spike = act_fun(mem1) # act_fun : approximation firing function
18.         spike2 = act_fun(mem2)
19.         return mem1, spike, spike-spike2
```

APPENDIX C

This part provides the code of three steps of training SNN mentioned in Chapter 4:

```
1.  class LIFNode(nn.Module):
2.      def __init__(self):
3.          super(LIFNode, self).__init__()
4.
5.      def forward(self, ops, x, mem, spike):
6.
7.          # step 1 T=1
8.          noise = torch.randn(mem.size(),device=device)
9.          maximum = torch.max(abs(noise))
10.         noise = noise/(2*maximum)+0.5
11.
12.         mem = noise + ops(x)
13.
14.         # step 2 T=N
15.         '''
16.         noise = torch.randn(mem.size(),device=device)
17.         maximum = torch.max(abs(noise))
18.         noise = noise/(2*maximum)+0.5
19.
20.         mem = noise + ops(x)
21.         '''
22.
23.         # step 3 T=N
24.         '''
25.         mean = torch.mean(mem * decay * (1. - spike))
26.         std = torch.std(mem * decay * (1. - spike), unbiased=False)
27.         mem = (mem * decay * (1. - spike)-mean)/(std+1e-17)
28.         maximum = torch.max(abs(mem))
29.
30.         noise = torch.randn(mem.size(),device=device)
31.         mem = mem/(4*maximum+1e-17)+0.5+noise/torch.max(abs(noise))*0.25
32.
33.         mem = mem + ops(x)
34.         '''
35.
36.         spike = act_fun(mem) # act_fun : approximation firing function
37.
38.
39.         return mem, spike
```

APPENDIX D

This part provides code of the mask SNN discussed in Chapter 5:

```
1.  # define approximate firing function
2.  class ActFun(torch.autograd.Function):
3.      @staticmethod
4.      def forward(ctx, input, thresh):
5.          ctx.save_for_backward(input)
6.          ctx.th = thresh
7.          return input.gt(thresh).float()
8.  
9.      @staticmethod
10.     def backward(ctx, grad_output):
11.         input, = ctx.saved_tensors
12.         grad_input = grad_output.clone()
13.         temp = alpha / 2 / (1 + (math.pi / 2 * alpha * (input-ctx.th)).pow_(2))
14.         return grad_input * temp.float(),None,None
15. 
16. act_fun = ActFun.apply
17. 
18. 
19. class IFNode(nn.Module):
20.     def __init__(self):
21.         super(IFNode, self).__init__()
22.         self.w = torch.nn.Parameter(torch.ones(1), requires_grad=True)
23.     def forward(self, x, thresh):
24.         spike = act_fun(x, thresh) # act_fun : approximation firing function
25.         return spike
26. 
27. class SNN(nn.Module):
28.     def __init__(self, T):
29.         super().__init__()
30.         self.T = T
31.         self.flatten = nn.Flatten()
32.         self.encoder = nn.Sequential(
33.             nn.Conv2d(1, 16, 3, padding=1),
34.             nn.BatchNorm2d(16),
35.             neuron.LIFNode(tau=1.2,surrogate_function=surrogate.ATan()),
36.             nn.MaxPool2d(2, stride=2),
37. 
38.             nn.Conv2d(16, 4, 3, padding=1),
39.             nn.BatchNorm2d(4),
40.             neuron.LIFNode(tau=1.2,surrogate_function=surrogate.ATan()),
41.             nn.MaxPool2d(2, stride=2),
42.         )
43. 
44.         self.decoder = nn.Sequential(
45.             nn.ConvTranspose2d(4, 16, 2, stride=2),
46.             nn.BatchNorm2d(16),
47.             neuron.LIFNode(tau=1.2,surrogate_function=surrogate.ATan()),
48. 
49.             nn.ConvTranspose2d(16, 1, 2, stride=2),
50.             nn.BatchNorm2d(1),
51. 
52.         )
53. 
54.         self.func = IFNode()
55. 
56.     def forward(self, x):
57.         dim = x.size()
58.         out_mem_counter = self.decoder(self.encoder(x))
59.         for t in range(1, self.T):
```

```
60.                out_mem_counter = out_mem_counter + self.decoder(self.encoder(x))
61.            out_mem_counter = self.flatten(out_mem_counter)
62.            a = out_mem_counter
63.            th = torch.tensor(np.percentile(a.cpu().detach().numpy(), 90, axis=1, keepd
    ims=True)).cuda()
64.            mask = self.func(out_mem_counter, th)
65.            mask = mask.view(-1,1,int(dim[-2]),int(dim[-1]))
66.
67.            return mask
```

APPENDIX E

This part provides the training and inference files for Chapter 6.

Training file:

```
1.  import numpy as np
2.  np.set_printoptions(threshold=np.inf)
3.  import matplotlib.cm as cmap
4.  # import tkinter
5.  # import matplotlib
6.  # matplotlib.use('TkAgg')
7.  
8.  import matplotlib.pyplot as plt
9.  import time
10. import os.path
11. import scipy
12. import pickle
13. from struct import unpack
14. from brian2 import *
15. import brian2 as b2
16. from brian2tools import *
17. 
18. #prefs.codegen.target = 'cython'
19. 
20. # specify the location of the data
21. data_path = ''
22. print(os.path.abspath(data_path))
23. 
24. #------------------------------------------------------------------------------
25. # functions
26. #------------------------------------------------------------------------------
27. def get_labeled_data(picklename, bTrain = True):
28.     """Read input-vector (image) and target class (label, 0-9) and return
29.        it as list of tuples.
30.     """
31.     if os.path.isfile('%s.pickle' % picklename):
32.         data = pickle.load(open('%s.pickle' % picklename,'rb'))
33. 
34.     else:
35.         # Open the images with gzip in read binary mode
36.         if bTrain:
37.             images = open(data_path + 'train-images.idx3-ubyte','rb')
38.             labels = open(data_path + 'train-labels.idx1-ubyte','rb')
39.         else:
40.             images = open(data_path + 'test-images6.idx3-ubyte','rb')
41.             labels = open(data_path + 'test-labels6.idx1-ubyte','rb')
42.         # Get metadata for images
43.         images.read(4)  # skip the magic_number
44.         number_of_images = unpack('>I', images.read(4))[0]
45.         print(number_of_images)
46.         rows = unpack('>I', images.read(4))[0]
47.         cols = unpack('>I', images.read(4))[0]
48.         # Get metadata for labels
49.         labels.read(4)  # skip the magic_number
50.         N = unpack('>I', labels.read(4))[0]
51. 
52.         if number_of_images != N:
53.             raise Exception('number of labels did not match the number of images')

54.         # Get the data
55.         x = np.zeros((N, rows, cols), dtype=np.uint8)  # Initialize numpy array
```

```
56.         y = np.zeros((N, 1), dtype=np.uint8)  # Initialize numpy array
57.         for i in range(N):
58.             if i % 1000 == 0:
59.                 print("i: %i" % i)
60.             x[i] = [[unpack('>B', images.read(1))[0] for unused_col in range(cols)]
    for unused_row in range(rows) ]
61.             y[i] = unpack('>B', labels.read(1))[0]
62. 
63.         data = {'x': x, 'y': y, 'rows': rows, 'cols': cols}
64.         pickle.dump(data, open("%s.pickle" % picklename, "wb"))
65.     return data
66. 
67. def get_matrix_from_file(fileName):
68.     offset = len(ending) + 4
69.     if fileName[-4-offset] == 'X':
70.         n_src = n_input
71.     else:
72.         if fileName[-3-offset]=='e':
73.             n_src = n_e
74.         else:
75.             n_src = n_i
76.     if fileName[-1-offset]=='e':
77.         n_tgt = n_e
78.     else:
79.         n_tgt = n_i
80.     readout = np.load(fileName)
81.     print(readout.shape, fileName)
82.     value_arr = np.zeros((n_src, n_tgt))
83.     if not readout.shape == (0,):
84.         value_arr[np.int32(readout[:,0]), np.int32(readout[:,1])] = readout[:,2]
85.     return value_arr
86. 
87. 
88. def save_connections(ending = ''):
89.     print('save connections')
90.     for connName in save_conns:
91.         conn = connections[connName]
92.         connListSparse = list(zip(conn.i, conn.j, conn.w))
93.         np.save(data_path + 'weights/' + connName + ending, connListSparse)
94. 
95. def save_theta(ending = ''):
96.     print('save theta')
97.     for pop_name in population_names:
98.         np.save(data_path + 'weights/theta_' + pop_name + ending, neuron_groups[pop
_name + 'e'].theta)
99. 
100.        def normalize_weights():
101.            for connName in connections:
102.                if connName[1] == 'e' and connName[3] == 'e':
103.                    len_source = len(connections[connName].source)
104.                    len_target = len(connections[connName].target)
105.                    connection = np.zeros((len_source, len_target))
106.                    connection[connections[connName].i, connections[connName].j] = c
onnections[connName].w
107.                    temp_conn = np.copy(connection)
108.                    colSums = np.sum(temp_conn, axis = 0)
109.                    colFactors = weight['ee_input']/colSums
110.                    for j in range(n_e):#
111.                        temp_conn[:,j] *= colFactors[j]
112.                    connections[connName].w = temp_conn[connections[connName].i, con
nections[connName].j]
113. 
114.        def get_2d_input_weights():
115.            name = 'XeAe'
116.            weight_matrix = np.zeros((n_input, n_e))
117.            n_e_sqrt = int(np.sqrt(n_e))
```

```
118.            n_in_sqrt = int(np.sqrt(n_input))
119.            num_values_col = n_e_sqrt*n_in_sqrt
120.            num_values_row = num_values_col
121.            rearranged_weights = np.zeros((num_values_col, num_values_row))
122.            connMatrix = np.zeros((n_input, n_e))
123.            connMatrix[connections[name].i, connections[name].j] = connections[name]
    .w
124.            weight_matrix = np.copy(connMatrix)
125.
126.            for i in range(n_e_sqrt):
127.                for j in range(n_e_sqrt):
128.                        rearranged_weights[i*n_in_sqrt : (i+1)*n_in_sqrt, j*n_in_sqr
    t : (j+1)*n_in_sqrt] = \
129.                            weight_matrix[:, i + j*n_e_sqrt].reshape((n_in_sqrt, n_i
    n_sqrt))
130.            return rearranged_weights
131.
132.
133.        def plot_2d_input_weights():
134.            name = 'XeAe'
135.            weights = get_2d_input_weights()
136.            fig = b2.figure(fig_num, figsize = (18, 18))
137.            im2 = b2.imshow(weights, interpolation = "nearest", vmin = 0, vmax = wma
    x_ee, cmap = cmap.get_cmap('hot_r'))
138.            b2.colorbar(im2)
139.            b2.title('weights of connection' + name)
140.            fig.canvas.draw()
141.            return im2, fig
142.
143.        def update_2d_input_weights(im, fig):
144.            weights = get_2d_input_weights()
145.            im.set_array(weights)
146.            fig.canvas.draw()
147.            return im
148.
149.        def get_current_performance(performance, current_example_num):
150.            current_evaluation = int(current_example_num/update_interval)
151.            start_num = current_example_num - update_interval
152.            end_num = current_example_num
153.            difference = outputNumbers[start_num:end_num, 0] - input_numbers[start_n
    um:end_num]
154.            correct = len(np.where(difference == 0)[0])
155.            performance[current_evaluation] = correct / float(update_interval) * 100
     #performance[current_evaluation] = correct / float(update_interval) * 100
156.            if (current_example_num + 1) == num_examples:
157.                performance[current_evaluation+1] = correct / float(update_interval)
     * 100
158.            return performance
159.
160.        def plot_performance(fig_num):
161.            num_evaluations = int(num_examples/update_interval) + 1
162.            time_steps = range(0, num_evaluations)
163.            performance = np.zeros(num_evaluations)
164.            fig = b2.figure(fig_num, figsize = (5, 5))
165.            fig_num += 1
166.            ax = fig.add_subplot(111)
167.            im2, = ax.plot(time_steps, performance) #my_cmap
168.            b2.ylim(ymax = 100)
169.            b2.title('Classification performance')
170.            fig.canvas.draw()
171.            return im2, performance, fig_num, fig
172.
173.        def update_performance_plot(im, performance, current_example_num, fig):
174.            performance = get_current_performance(performance, current_example_num)

175.            im.set_ydata(performance)
```

```
176.            fig.canvas.draw()
177.            return im, performance
178.
179.        def get_recognized_number_ranking(assignments, spike_rates):
180.            summed_rates = [0] * 8
181.            num_assignments = [0] * 8
182.            for i in range(8):
183.                num_assignments[i] = len(np.where(assignments == i)[0])
184.                if num_assignments[i] > 0:
185.                    summed_rates[i] = np.sum(spike_rates[assignments == i]) / num_as
    signments[i]
186.            return np.argsort(summed_rates)[::-1]
187.
188.        def get_new_assignments(result_monitor, input_numbers):
189.            assignments = np.zeros(n_e)
190.            input_nums = np.asarray(input_numbers)
191.            maximum_rate = [0] * n_e
192.            rate = [0] * n_e
193.            for j in range(8):
194.                num_assignments = len(np.where(input_nums == j)[0])
195.                if num_assignments > 0:
196.                    rate = np.sum(result_monitor[input_nums == j], axis = 0) / num_a
    ssignments
197.
198.                for i in range(n_e):
199.                    if rate[i] > maximum_rate[i]:
200.                        maximum_rate[i] = rate[i]
201.                        assignments[i] = j
202.            return assignments
203.
204.
205.        #-----------------------------------------------------------------------------
    ---
206.        # load data
207.        #-----------------------------------------------------------------------------
    ---
208.        start = time.time()
209.        training = get_labeled_data(data_path + 'training')
210.        print(len(training['x']))
211.        end = time.time()
212.        print('time needed to load training set:', end - start)
213.
214.        start = time.time()
215.        testing = get_labeled_data(data_path + 'testing', bTrain = False)
216.        print(len(testing['x']))
217.        end = time.time()
218.        print('time needed to load test set:', end - start)
219.
220.
221.        #-----------------------------------------------------------------------------
    ---
222.        # set parameters and equations
223.        #-----------------------------------------------------------------------------
    ---
224.        test_mode = True
225.
226.        np.random.seed(0)
227.        data_path = './'
228.        if test_mode:
229.            weight_path = data_path + 'weights/'
230.            num_examples = 24 * 1
231.            use_testing_set = True
232.            do_plot_performance = False
233.            record_spikes = True
234.            ee_STDP_on = False
235.            update_interval = num_examples
```

```
236.         else:
237.             weight_path = data_path + 'random/'
238.             num_examples = 2696 * 16
239.             use_testing_set = False
240.             do_plot_performance = True
241.             if num_examples <= 2696:
242.                 record_spikes = True
243.             else:
244.                 record_spikes = True
245.             ee_STDP_on = True
246. 
247. 
248.         ending = ''
249.         n_input = 784
250.         n_e = 400
251.         n_i = n_e
252.         single_example_time =   0.35 * b2.second #
253.         resting_time = 0.15 * b2.second
254.         runtime = num_examples * (single_example_time + resting_time)
255.         if num_examples <= 24:
256.             update_interval = num_examples
257.             weight_update_interval = 20
258.         else:
259.             update_interval = 2696
260.             weight_update_interval = 100
261.         if num_examples <= 2696:
262.             save_connections_interval = 2696
263.         else:
264.             save_connections_interval = 2696
265.             update_interval = 2696
266. 
267.         v_rest_e = -65. * b2.mV
268.         v_rest_i = -60. * b2.mV
269.         v_reset_e = -65. * b2.mV
270.         v_reset_i = -45. * b2.mV
271.         v_thresh_e = -52. * b2.mV
272.         v_thresh_i = -40. * b2.mV
273.         refrac_e = 5. * b2.ms
274.         refrac_i = 2. * b2.ms
275. 
276.         weight = {}
277.         delay = {}
278.         input_population_names = ['X']
279.         population_names = ['A']
280.         input_connection_names = ['XA']
281.         save_conns = ['XeAe']
282.         input_conn_names = ['ee_input']
283.         recurrent_conn_names = ['ei', 'ie']
284.         weight['ee_input'] = 78.
285.         delay['ee_input'] = (0*b2.ms,10*b2.ms)
286.         delay['ei_input'] = (0*b2.ms,5*b2.ms)
287.         input_intensity = 2.
288.         start_input_intensity = input_intensity
289. 
290.         tc_pre_ee = 20*b2.ms
291.         tc_post_1_ee = 20*b2.ms
292.         tc_post_2_ee = 40*b2.ms
293.         nu_ee_pre =  0.0001      # learning rate
294.         nu_ee_post = 0.01       # learning rate
295.         wmax_ee = 1.0
296.         exp_ee_pre = 0.2
297.         exp_ee_post = exp_ee_pre
298.         STDP_offset = 0.4
299. 
300.         if test_mode:
301.             scr_e = 'v = v_reset_e; timer = 0*ms'
```

```
302.        else:
303.            tc_theta = 1e7 * b2.ms
304.            theta_plus_e = 0.05 * b2.mV
305.            scr_e = 'v = v_reset_e; theta += theta_plus_e; timer = 0*ms'
306.        offset = 20.0*b2.mV
307.        v_thresh_e_str = '(v>(theta - offset + v_thresh_e)) and (timer>refrac_e)'
308.        v_thresh_i_str = 'v>v_thresh_i'
309.        v_reset_i_str = 'v=v_reset_i'
310.
311.
312.        neuron_eqs_e = '''''
313.                dv/dt = ((v_rest_e - v) + (I_synE+I_synI) / nS) / (100*ms)  : volt (
    unless refractory)
314.                I_synE = ge * nS *         -v                           : amp
315.                I_synI = gi * nS * (-100.*mV-v)                          : amp
316.                dge/dt = -ge/(1.0*ms)                                   : 1
317.                dgi/dt = -gi/(2.0*ms)                                  : 1
318.                '''
319.        if test_mode:
320.            neuron_eqs_e += '\n  theta      :volt'
321.        else:
322.            neuron_eqs_e += '\n  dtheta/dt = -theta / (tc_theta)  : volt'
323.        neuron_eqs_e += '\n  dtimer/dt = 0.1  : second'
324.
325.        neuron_eqs_i = '''''
326.                dv/dt = ((v_rest_i - v) + (I_synE+I_synI) / nS) / (10*ms)  : volt (u
    nless refractory)
327.                I_synE = ge * nS *         -v                           : amp
328.                I_synI = gi * nS * (-85.*mV-v)                          : amp
329.                dge/dt = -ge/(1.0*ms)                                   : 1
330.                dgi/dt = -gi/(2.0*ms)                                  : 1
331.                '''
332.        eqs_stdp_ee = '''''
333.                        post2before                        : 1
334.                        dpre/dt    =    -pre/(tc_pre_ee)         : 1 (event-driven)
335.                        dpost1/dt  = -post1/(tc_post_1_ee)     : 1 (event-driven)
336.                        dpost2/dt  = -post2/(tc_post_2_ee)     : 1 (event-driven)
337.                    '''
338.        eqs_stdp_pre_ee = 'pre = 1.; w = clip(w + nu_ee_pre * post1, 0, wmax_ee)'
339.        eqs_stdp_post_ee = 'post2before = post2; w = clip(w + nu_ee_post * pre * pos
    t2before, 0, wmax_ee); post1 = 1.; post2 = 1.'
340.
341.        '''''
342.        eqs_stdp_ee = '''
343.                         dpre/dt = -pre/(tc_pre_ee)  : 1 (event-driven)
344.                         dpost1/dt  = -post1/(tc_post_1_ee) : 1 (event-driven)'''''
345.        eqs_stdp_pre_ee = 'pre = 1.; w = clip(w + nu_ee_pre * post1, 0, wmax_ee)'
346.        eqs_stdp_post_ee = 'w = clip(w + nu_ee_post * pre , 0, wmax_ee); post1 = 1.'

347.        '''
348.
349.        b2.ion()
350.        fig_num = 1
351.        neuron_groups = {}
352.        input_groups = {}
353.        connections = {}
354.        rate_monitors = {}
355.        spike_monitors = {}
356.        spike_counters = {}
357.        result_monitor = np.zeros((update_interval,n_e))
358.
359.        neuron_groups['e'] = b2.NeuronGroup(n_e*len(population_names), neuron_eqs_e,
     threshold= v_thresh_e_str, refractory= refrac_e, reset= scr_e, method='euler')
360.        neuron_groups['i'] = b2.NeuronGroup(n_i*len(population_names), neuron_eqs_i,
     threshold= v_thresh_i_str, refractory= refrac_i, reset= v_reset_i_str, method='eul
    er')
```

```
361.
362.
363.        #-----------------------------------------------------------------------
---
364.        # create network population and recurrent connections
365.        #-----------------------------------------------------------------------
---
366.        for subgroup_n, name in enumerate(population_names):
367.            print('create neuron group', name)
368.
369.            neuron_groups[name+'e'] = neuron_groups['e'][subgroup_n*n_e:(subgroup_n+
1)*n_e]
370.            neuron_groups[name+'i'] = neuron_groups['i'][subgroup_n*n_i:(subgroup_n+
1)*n_e]
371.
372.            neuron_groups[name+'e'].v = v_rest_e - 40. * b2.mV
373.            neuron_groups[name+'i'].v = v_rest_i - 40. * b2.mV
374.            if test_mode or weight_path[-8:] == 'weights/':
375.                neuron_groups['e'].theta = np.load(weight_path + 'theta_' + name + e
nding + '.npy') * b2.volt
376.                #neuron_groups['e'].theta = np.load(weight_path + 'theta_A2696.npy')
 * b2.volt
377.            else:
378.                neuron_groups['e'].theta = np.load(weight_path + 'theta_' + name + e
nding + '.npy') * b2.volt
379.                #neuron_groups['e'].theta = np.ones((n_e)) * 20.0*b2.mV
380.
381.            print('create recurrent connections')
382.            for conn_type in recurrent_conn_names:
383.                connName = name+conn_type[0]+name+conn_type[1]
384.                weightMatrix = get_matrix_from_file(weight_path + '../random/' + con
nName + ending + '.npy')
385.                #print(weight_path + '../random/' + connName + ending + '.npy')
386.                model = 'w : 1'
387.                pre = 'g%s_post += w' % conn_type[0]
388.                post = ''
389.                if ee_STDP_on:
390.                    if 'ee' in recurrent_conn_names:
391.                        model += eqs_stdp_ee
392.                        pre += '; ' + eqs_stdp_pre_ee
393.                        post = eqs_stdp_post_ee
394.                connections[connName] = b2.Synapses(neuron_groups[connName[0:2]], ne
uron_groups[connName[2:4]],
395.                                                        model=model, on_pre=pre,
 on_post=post)
396.                connections[connName].connect(True) # all-to-all connection
397.                connections[connName].w = weightMatrix[connections[connName].i, conn
ections[connName].j]
398.            #print(connections['AiAe'].w )
399.
400.            print('create monitors for', name)
401.            rate_monitors[name+'e'] = b2.PopulationRateMonitor(neuron_groups[name+'e
'])
402.            rate_monitors[name+'i'] = b2.PopulationRateMonitor(neuron_groups[name+'i
'])
403.            spike_counters[name+'e'] = b2.SpikeMonitor(neuron_groups[name+'e'])
404.
405.            if record_spikes:
406.                spike_monitors[name+'e'] = b2.SpikeMonitor(neuron_groups[name+'e'])

407.                spike_monitors[name+'i'] = b2.SpikeMonitor(neuron_groups[name+'i'])

408.
409.
410.        #-----------------------------------------------------------------------
---
```

```
411.        # create input population and connections from input populations
412.        #-----------------------------------------------------------------------------
---
413.        pop_values = [0,0,0]
414.        for i,name in enumerate(input_population_names):
415.            input_groups[name+'e'] = b2.PoissonGroup(n_input, 0*Hz)
416.            rate_monitors[name+'e'] = b2.PopulationRateMonitor(input_groups[name+'e'
])
417.
418.        for name in input_connection_names:
419.            print('create connections between', name[0], 'and', name[1])
420.            for connType in input_conn_names:
421.                connName = name[0] + connType[0] + name[1] + connType[1]
422.                weightMatrix = get_matrix_from_file(weight_path + connName + ending
+ '.npy')
423.                #weightMatrix = get_matrix_from_file(weight_path + 'XeAe13480.npy')

424.                #print(weight_path + connName + ending + '.npy112')
425.                model = 'w : 1'
426.                pre = 'g%s_post += w' % connType[0]
427.                post = ''
428.                if ee_STDP_on:
429.                    print('create STDP for connection', name[0]+'e'+name[1]+'e')
430.                    model += eqs_stdp_ee
431.                    pre += '; ' + eqs_stdp_pre_ee
432.                    post = eqs_stdp_post_ee
433.
434.                connections[connName] = b2.Synapses(input_groups[connName[0:2]], neu
ron_groups[connName[2:4]],
435.                                                            model=model, on_pre=pre,
 on_post=post)
436.                minDelay = delay[connType][0]
437.                maxDelay = delay[connType][1]
438.                deltaDelay = maxDelay - minDelay
439.                # TODO: test this
440.                connections[connName].connect(True) # all-to-all connection
441.                connections[connName].delay = 'minDelay + rand() * deltaDelay'
442.                connections[connName].w = weightMatrix[connections[connName].i, conn
ections[connName].j]
443.
444.
445.        #-----------------------------------------------------------------------------
---
446.        # run the simulation and set inputs
447.        #-----------------------------------------------------------------------------
---
448.
449.        net = Network()
450.        for obj_list in [neuron_groups, input_groups, connections, rate_monitors,
451.                spike_monitors, spike_counters]:
452.            for key in obj_list:
453.                net.add(obj_list[key])
454.
455.        previous_spike_count = np.zeros(n_e)
456.        assignments = np.zeros(n_e)
457.        input_numbers = [0] * num_examples
458.        outputNumbers = np.zeros((num_examples, 8))
459.        if not test_mode:
460.            input_weight_monitor, fig_weights = plot_2d_input_weights()
461.            fig_num += 1
462.        if do_plot_performance:
463.            performance_monitor, performance, fig_num, fig_performance = plot_perfor
mance(fig_num)
464.        for i,name in enumerate(input_population_names):
465.            input_groups[name+'e'].rates = 0 * Hz
466.        net.run(0*second)
```

```
467.        j = 0
468.        while j < (int(num_examples)):
469.            print(j,num_examples)
470.            if test_mode:
471.                if use_testing_set:
472.                    spike_rates = testing['x'][j%24,:,:].reshape((n_input)) / 8. *
    input_intensity
473.                else:
474.                    spike_rates = training['x'][j%2696,:,:].reshape((n_input)) / 8.
    *  input_intensity
475.
476.            else:
477.                normalize_weights()
478.                spike_rates = training['x'][j%2696,:,:].reshape((n_input)) / 8.  *  i
    nput_intensity
479.            input_groups['Xe'].rates = spike_rates * Hz
480.        #     print('run number:', j+1, 'of', int(num_examples))
481.            net.run(single_example_time, report='text')
482.            print(neuron_groups['Ae'].v[20])
483.            #print(spike_rates)
484.            if j % update_interval == 0 and j > 0:
485.                assignments = get_new_assignments(result_monitor[:], input_numbers[j
    -update_interval : j])
486.            if j % weight_update_interval == 0 and not test_mode:
487.                update_2d_input_weights(input_weight_monitor, fig_weights)
488.            if j % save_connections_interval == 0 and j > 0 and not test_mode:
489.                save_connections(str(j))
490.                save_theta(str(j))
491.
492.            current_spike_count = np.asarray(spike_counters['Ae'].count[:]) - previo
    us_spike_count
493.            previous_spike_count = np.copy(spike_counters['Ae'].count[:])
494.            if np.sum(current_spike_count) < 5:
495.                input_intensity += 1
496.                for i,name in enumerate(input_population_names):
497.                    input_groups[name+'e'].rates = 0 * Hz
498.                net.run(resting_time)
499.            else:
500.                result_monitor[j%update_interval,:] = current_spike_count
501.                if test_mode and use_testing_set:
502.                    input_numbers[j] = testing['y'][j%24][0]
503.                else:
504.                    input_numbers[j] = training['y'][j%2696][0]
505.                outputNumbers[j,:] = get_recognized_number_ranking(assignments, resu
    lt_monitor[j%update_interval,:])
506.                if j % 100 == 0 and j > 0:
507.                    print('runs done:', j, 'of', int(num_examples))
508.                if j % update_interval == 0 and j > 0: #update_interval
509.                    if do_plot_performance:
510.                        unused, performance = update_performance_plot(performance_mo
    nitor, performance, j, fig_performance)
511.                        print('Classification performance', performance[:int(j/updat
    e_interval+1)])
512.                if j % (num_examples-1) == 0 and j > 0: #update_interval
513.                    if do_plot_performance:
514.                        unused, performance = update_performance_plot(performance_mo
    nitor, performance, j, fig_performance)
515.                        print('Classification performance', performance[:int(j/updat
    e_interval+2)])
516.                for i,name in enumerate(input_population_names):
517.                    input_groups[name+'e'].rates = 0 * Hz
518.                if (j+1)%1000==0:
519.                    #plt.figure(j+10)
520.                    plot_2d_input_weights()
521.                    plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\pictu
    re2\Image'+str(j+1))
```

```
522.                     plt.clf()
523.                 net.run(resting_time)
524.                 input_intensity = start_input_intensity
525.                 j += 1
526.
527.
528.
529.
530.         #------------------------------------------------------------------------
    ---
531.         # save results
532.         #------------------------------------------------------------------------
    ---
533.         print('save results')
534.         if not test_mode:
535.             save_theta()
536.         if not test_mode:
537.             save_connections()
538.             np.save(data_path + 'activity/resultPopVecs' + str(num_examples), result
    _monitor)
539.             np.save(data_path + 'activity/inputNumbers' + str(num_examples), input_n
    umbers)
540.         else:
541.             np.save(data_path + 'activity/resultPopVecs' + str(num_examples) +'', re
    sult_monitor)
542.             np.save(data_path + 'activity/inputNumbers' + str(num_examples) +'', inp
    ut_numbers)
543.
544.
545.
546.         #------------------------------------------------------------------------
    ---
547.         # plot results
548.         #------------------------------------------------------------------------
    ---
549.         if rate_monitors:
550.             b2.figure(fig_num)
551.             fig_num += 1
552.             for i, name in enumerate(rate_monitors):
553.                 b2.subplot(len(rate_monitors), 1, 1+i)
554.                 b2.plot(rate_monitors[name].t/b2.second, rate_monitors[name].rate, '
    .')
555.                 b2.title('Rates of population ' + name)
556.                 plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\picture2\
    Image'+str(1))
557.
558.         if spike_monitors:
559.             b2.figure(fig_num)
560.
561.             fig_num += 1
562.             for i, name in enumerate(spike_monitors):
563.                 b2.subplot(len(spike_monitors), 1, 1+i)
564.                 b2.plot(spike_monitors[name].t/b2.ms, spike_monitors[name].i, '.')
565.                 b2.title('Spikes of population ' + name)
566.                 plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\picture2\
    Image'+str(2))
567.
568.         if spike_counters:
569.             b2.figure(fig_num)
570.             fig_num += 1
571.             b2.plot(spike_monitors['Ae'].count[:])
572.             b2.title('Spike count of population Ae')
573.             plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\picture2\Imag
    e'+str(3))
574.
575.
```

```
576.
577.
578.
579.        plot_2d_input_weights()
580.        plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\picture2\Image'+s
    tr(4))
581.
582.        plt.figure(5)
583.
584.        subplot(3,1,1)
585.
586.        brian_plot(connections['XeAe'].w)
587.        subplot(3,1,2)
588.
589.        brian_plot(connections['AeAi'].w)
590.
591.        subplot(3,1,3)
592.
593.        brian_plot(connections['AiAe'].w)
594.        plt.savefig('E:\Codeproject\TactileSnn_new\Tactiletrain400\picture2\Image'+s
    tr(5))
595.
596.
597.
598.        b2.ioff()
599.        b2.show()
```

Inference file:

```
1.  import numpy as np
2.  import matplotlib
3.  import matplotlib.cm as cmap
4.  import time
5.  import os.path
6.  import scipy
7.  import pickle
8.  from struct import unpack
9.  from brian2 import *
10.
11.
12. #------------------------------------------------------------------------------
13. # functions
14. #-----------------------------------------------------------------------------
    -
15. def get_labeled_data(picklename, bTrain = True):
16.     """Read input-vector (image) and target class (label, 0-9) and return
17.        it as list of tuples.
18.     """
19.     if os.path.isfile('%s.pickle' % picklename):
20.         data = pickle.load(open('%s.pickle' % picklename,'rb'))
21.     else:
22.         # Open the images with gzip in read binary mode
23.         if bTrain:
24.             images = open(data_path + 'train-images.idx3-ubyte','rb')
25.             labels = open(data_path + 'train-labels.idx1-ubyte','rb')
26.         else:
27.             images = open(data_path + 't10k-images.idx3-ubyte','rb')
28.             labels = open(data_path + 't10k-labels.idx1-ubyte','rb')
29.         # Get metadata for images
30.         images.read(4)  # skip the magic_number
31.         number_of_images = unpack('>I', images.read(4))[0]
```

```
32.         rows = unpack('>I', images.read(4))[0]
33.         cols = unpack('>I', images.read(4))[0]
34.         # Get metadata for labels
35.         labels.read(4)  # skip the magic_number
36.         N = unpack('>I', labels.read(4))[0]
37.         if number_of_images != N:
38.             raise Exception('number of labels did not match the number of images')

39.         # Get the data
40.         x = np.zeros((N, rows, cols), dtype=np.uint8)  # Initialize numpy array
41.         y = np.zeros((N, 1), dtype=np.uint8)  # Initialize numpy array
42.         for i in range(N):
43.             if i % 100 == 0:
44.                 print("i: %i" % i)
45.             x[i] = [[unpack('>B', images.read(1))[0] for unused_col in range(cols)]
     for unused_row in range(rows) ]
46.             y[i] = unpack('>B', labels.read(1))[0]
47.         data = {'x': x, 'y': y, 'rows': rows, 'cols': cols}
48.         pickle.dump(data, open("%s.pickle" % picklename, "wb"))
49.     return data
50.
51. def get_recognized_number_ranking(assignments, spike_rates):
52.     summed_rates = [0] * 8
53.     num_assignments = [0] * 8
54.     for i in range(8):
55.         num_assignments[i] = len(np.where(assignments == i)[0])
56.         if num_assignments[i] > 0:
57.             summed_rates[i] = np.sum(spike_rates[assignments == i]) / num_assignmen
    ts[i]
58.     return np.argsort(summed_rates)[::-1]
59.
60. def get_new_assignments(result_monitor, input_numbers):
61.     print(result_monitor.shape)
62.     assignments = np.ones(n_e) * -1 # initialize them as not assigned
63.     input_nums = np.asarray(input_numbers)
64.     maximum_rate = [0] * n_e
65.     rate = [0] * n_e
66.     for j in range(8):
67.         num_inputs = len(np.where(input_nums == j)[0])
68.         if num_inputs > 0:
69.             rate = np.sum(result_monitor[input_nums == j], axis = 0) / num_inputs
70.         for i in range(n_e):
71.             if rate[i] > maximum_rate[i]:
72.                 maximum_rate[i] = rate[i]
73.                 assignments[i] = j
74.     return assignments
75.
76. data_path = './'
77. data_path = './activity/'
78. training_ending = '2696'
79. testing_ending = '24'
80. start_time_training = 0
81. end_time_training = int(training_ending)
82. start_time_testing = 0
83. end_time_testing = int(testing_ending)
84.
85. n_e = 400
86. n_input = 784
87. ending = ''
88.
89. print('load data')
90. training = get_labeled_data(data_path + 'training')
91. testing = get_labeled_data(data_path + 'testing', bTrain = False)
92.
93. print('load results')
```

```
94. training_result_monitor = np.load(data_path + 'resultPopVecs' + training_ending + e
    nding + '.npy')
95. training_input_numbers = np.load(data_path + 'inputNumbers' + training_ending + '.n
    py')
96. testing_result_monitor = np.load(data_path + 'resultPopVecs' + testing_ending + '.n
    py')
97. testing_input_numbers = np.load(data_path + 'inputNumbers' + testing_ending + '.npy
    ')
98. print(training_result_monitor.shape)
99.
100.        print('get assignments')
101.        test_results = np.zeros((8, end_time_testing-start_time_testing))
102.        test_results_max = np.zeros((8, end_time_testing-start_time_testing))
103.        test_results_top = np.zeros((8, end_time_testing-start_time_testing))
104.        test_results_fixed = np.zeros((8, end_time_testing-start_time_testing))
105.        assignments = get_new_assignments(training_result_monitor[start_time_trainin
    g:end_time_training],
106.                                          training_input_numbers[start_time_training
    :end_time_training])
107.        print(assignments)
108.        counter = 0
109.        num_tests = end_time_testing / 24
110.        sum_accurracy = [0] * int(num_tests)
111.        while (counter < num_tests):
112.            end_time = min(end_time_testing, 24*(counter+1))
113.            start_time = 24*counter
114.            test_results = np.zeros((8, end_time-start_time))
115.            print('calculate accuracy for sum')
116.            for i in range(end_time - start_time):
117.                test_results[:,i] = get_recognized_number_ranking(assignments,
118.                                                                  testing_result_mon
    itor[i+start_time,:])
119.            difference = test_results[0,:] - testing_input_numbers[start_time:end_ti
    me]
120.            for i in np.where(difference != 0)[0]:
121.                print(testing_input_numbers[i],test_results[0,i])
122.
123.            correct = len(np.where(difference == 0)[0])
124.            incorrect = np.where(difference != 0)[0]
125.            sum_accurracy[counter] = correct #/float(end_time-start_time) * 100
126.            print('Sum response - accuracy: ', sum_accurracy[counter], ' num incorre
    ct: ', len(incorrect))
127.            counter += 1
128.        print('Sum response - accuracy --> mean: ', np.mean(sum_accurracy),  '-
    -> standard deviation: ', np.std(sum_accurracy))
129.
130.
131.        show()
```