Energy-Efficient Models for High-Dimensional Spike Train Classification using Sparse Spiking Neural Networks

Hang Yin
Worcester Polytechnic Institute
Worcester, USA
hyin@wpi.edu

John Lee
Worcester Polytechnic Institute
Worcester, USA
jtlee@wpi.edu

Xiangnan Kong
Worcester Polytechnic Institute
Worcester, USA
xkong@wpi.edu

Thomas Hartvigsen
Worcester Polytechnic Institute
Worcester, USA
twhartvigsen@wpi.edu

Sihong Xie
Lehigh University
Bethlehem, USA
six316@lehigh.edu

ABSTRACT

Spike train classification is an important problem in many areas such as healthcare and mobile sensing, where each spike train is a high-dimensional time series of binary values. Conventional research on spike train classification mainly focus on developing Spiking Neural Networks (SNNs) under resource-sufficient settings (e.g., on GPU servers). The neurons of the SNNs are usually densely connected in each layer. However, in many real-world applications, we often need to deploy the SNN models on resource-constrained platforms (e.g., mobile devices) to analyze high-dimensional spike train data. The high resource requirement of the densely-connected SNNs can make them hard to deploy on mobile devices. In this paper, we study the problem of energy-efficient SNNs with sparsely-connected neurons. We propose an SNN model with sparse spatio-temporal coding. Our solution is based on the re-parameterization of weights in an SNN and the application of sparsity regularization during optimization. We compare our work with the state-of-the-art SNNs and demonstrate that our sparse SNNs achieve significantly better computational efficiency on both neuromorphic and standard datasets with comparable classification accuracy. Furthermore, compared with densely-connected SNNs, we show that our method has a better capability of generalization on small-size datasets through extensive experiments.

CCS CONCEPTS

• **Information systems** → **Data mining**; • **Computer systems organization** → *Neural networks*; • **Computing methodologies** → *Supervised learning*.

KEYWORDS

spiking neural networks, supervised learning, spatio-temporal coding, sparsity, hard-concrete distribution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD '21, August 14–18, 2021, Virtual Event, Singapore.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467252

ACM Reference Format:
Hang Yin, John Lee, Xiangnan Kong, Thomas Hartvigsen, and Sihong Xie. 2021. Energy-Efficient Models for High-Dimensional Spike Train Classification using Sparse Spiking Neural Networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3447548.3467252

1 INTRODUCTION

Motivation. Our brains have about a hundred billion neurons that fire signals to communicate with each other all the time. Each signal is electrochemical in nature and is referred to as a spike, or an action potential. The most popular way to think of spike *trains* is as a digital sequence of events: 1 for a spike, and 0 for no spike. Such spike trains arise during physical sensory stimuli such as vision and motion, or abstract stimuli such as memory. Recently, spike train classification has attracted much attention in the field of data mining [17, 20, 25, 26]. Unlike tradition classification, classifying spike trains is a task with sequences of spikes as both inputs and outputs. By assuming that all spikes are discrete characteristic events, the processing of information is reduced to the timing and counting of said spikes. Designing machine learning algorithms for spike train classification is very important in many high-impact fields such as sensor systems for disease diagnosis and human activity monitoring.

Knowledge Gap. Spiking Neural Network (SNN) show great potential for dealing with spike train classification [21–23, 25, 26]. Originally proposed to imitate biological information processing [9], the neurons transfer information between one another via spike trains. Unlike Recurrent Neural Networks (RNN), which use continuous value as inputs and outputs, SNNs take sparse spike trains as inputs and outputs, building large-scale neural networks with far less energy and memory on neuromorphic hardware systems, which operate on principles that are fundamentally different from standard digital computers. Thus, SNNs are clear candidates for spike train classification.

However, opportunities are always accompanied by challenges. Due to significant advances in miniaturization of sensor systems, more and more smart devices such as wearable sensors and smart phones for elderly care and aerial robots appear around us, which can produce high-dimensional data in the form of spike trains. These devices require high quality pattern recognition to meet

Figure 1: An example of energy-e cient spike train classi - cation problem.

their design requirements. At the same time, they are often limited by available energy and thus low computational cost is required during inference. As illustrated in Figure 1, wearable devices incorporated with varieties of motion sensors are used to monitor di erent physical conditions of seniors for identifying their body conditions. They generate data that cover massive measurements, including heart rate (HR), blood pressure (BP), and oxygen saturation (SpO2), among others, and are expected to be collected by smart devices subject to limited power.To run SNNs e ciently on such high-dimensional data, we need to ensure both high classi cation accuracyand low computational cost during inference.

Regarding computational cost, however, modern SNNs [25, 26] perform many unnecessary computations due to their dense network architectures. These unnecessary computations are caused by weak connections between neurons. Weak connections play a limited role in model performance during inference, as shown in the example in Figure 1. On one hand, inferring Falling needs activity-related signals given by a person's intertial measurement units (IMU), HR, and BP. On the other hand, inferring Heart Disease doesn't consider signals from IMUs, but needs BP, SpO2 and could use more measurements from their photoplethysmograms. As a result, current SNNs are still not suitable for spike train classi cation, especially when the data is high dimensional and comes from devices with limited power.

Challenges. In this paper, we propose an energy-e cient method for high-dimensional spike train classi cation. To solve this problem, there are two main challenges:

Sparse SNN vs Sparse RNNs. Sparsi cation techniques have been employed in RNNs [18] to reduce computational costs. However, RNNs model sequences via continuous values, and are outmatched by SNNs for spike train classi cation. By sparsifying SNNs, we can avoid unnecessary computations caused by weak connections between neurons. A sparsi ed model has far fewer non-zero parameters and so performs fewer computations during inference, making it more power e cient. Thus, instead of using sparse RNNs, we must nd a way to sparsify the network structure of SNNs.

Sparsifying Inputs vs Networks. SNNs have been accelerated by either sparsifying the inputs [5] or using stochastic computing [1]. However, by only focusing on spike rate, as opposed to spike timing, they disregard a major component of the problem. A successful method must consider both rate and timing together to successfully perform high-dimensional spike train classi cation.

Figure 2: Comparison of the key di erences between SNNs [21, 25], M-SNNs [5], and proposed sparse SNN.

Proposed Method. Inspired by the success of Arti cial Neural Networks (ANN) with a sparse structure [7, 13], we propose an SNN model with sparse spatio-temporal coding. We reparameterize the connection between each neuron in an SNN by multiplying each original weight by a binary "gate". Each gate is considered to be a Bernoulli random variable. As a result, our proposed approach allows each neuron in the SNN to consider the necessity of coming into contact with each neuron in the next layer. Therefore, it allows us to penalize the possibility of each gate for being di erent nonzero with no further restrictions, thereby pruning weak links. This reduces the overall computational cost and adds the bene t of regularization, reducing over tting. We show through empirical evaluation on multiple real-world datasets that, compared to baselines, the sparse SNN we propose greatly speeds up computation while incurring only a negligible deterioration in classi cation performance. Meanwhile, we also show improved generalizability by varying the size of the training set.

Contributions . Our contributions in this paper can be summarized as follows:

- We de ne the problem of spike train classi cation, which is very important for smart devices with limited energy.
- We propose a sparse SNN for high-dimensional spike train classi cation to be performed on energy-limited smart devices.
- We demonstrate that our model outperforms recent state-of-the-art alternatives by achieving lower computational cost when tested on both neuromorphic as well as standard datasets with very negligible degradation in classi cation performance.
- We also demonstrate that our method has excellent generalization capability on small datasets.

2 RELATED WORK

In spike train classi cation, indirect" learning methods, such as ANN-to-SNN conversion [4, 6, 8, 15], have been proposed to high

dimensional inputs. These are indirect learning methods because a regular non-spiking ANN (e.g., a multi-layer perceptron) is initially used during the training phase. At inference-time, the trained model is then converted to an SNN. However, there are several disadvantages associated with such indirect training. First, it doesn't align well with how an SNN operates. In ANNs, it does not matter if activations are negative, but ring rates in SNNs are always positive. Furthermore, many limiting constraints are typically added while training the ANN models. These include not using bias terms, only supporting average pooling, and only using ReLU activation functions.

In response, methods for directly training an SNN have recently been proposed [12, 21, 26]. These approaches are mainly based on conventional gradient descent. Most notably, di erent from previous techniques based only on spatial back-propagation [10, 12], SNNs trained directly using back-propagation in both the spatial as well as the temporal domains [21, 26] have achieved state-of-the-art accuracy on the MNIST and N-MNIST datasets. However, although these methods perform better than the others described above on many real-world datasets, from the perspective of computational e ciency, they are still far from power-e cient in solving high-dimensional spike trains classi cation. Therefore there have been some recently-proposed power-e cient SNNs [5]. [5] aims to enforce more neurons silence by making input spikes of each neuron sparser. [1] introduces a stochastic SNN by exploiting the bene ts of stochastic computing to generate input spike trains and reduce the connection complexity. However, both of them are only applicable to standard datasets, but not to neuromorphic datasets

3 PRELIMINARY

To describe the SNN models with a sparse structure, we rst introduce the baseline framework for SNNs, as proposed in [26]. We begin by describing the simplest possible SNN, one which comprises a single neuron with one input entry. This neuron is a recurrent unit that is a ected by the current input, and the previous input and output. For each timestep C, it combines the current input with the previous input and output to compute a new value. This value can be referred to as the *membrane potential* in biological neural network. If the membrane is greater than a threshold, the neuron res and outputs 1 to indicate a spike, otherwise, it outputs 0 to indicate silence. Therefore, for each timestep C, the membrane and output are expressed as follows:

$$D_C = g D_{C-1}(1 - I_{C-1}) + F G_C, \quad (1)$$

$$I_C = \Theta(D_C - \vartheta) \quad (2)$$

where we write D_C, G_C, and I_C to denote the membrane potential, input, and output of the neuron on timestep C, respectively. $g \in (0, 1)$ is the time decay constant hyperparameter, and F and 1 are the connection and bias between input and this neuron, respectively. $\Theta(\cdot)$ is the step function, which satis es $\Theta(x) = 0$ when $x < 0$, otherwise $\Theta(x) = 1$.

The SNN expressed in Equations 1-2 mimics natural neural networks more closely than a traditional ANN. In this way, we represent a neuron as the parallel combination of a "leaky" resistor and a capacitor. The second term of the r.h.s. of Equation 1 is used as external current input to charge up the capacitor to update the potential D_C. If the neuron emits a spike ($I_C = 1$) at timestep C, the capacitor discharges to a resting potential (which we x at zero throughout this paper) by using the rst term in Equation 1.

An SNN is built by hooking together many of these simple neurons , so that the output of a neuron can be the input of another. We let $D_8^{C,=}$ and $I_8^{C,=}$ denote the membrane and output of neuron 8 in layer $=$ at timestep C. The network has parameters $W = \{W^1, \cdots, W^{\#-1}\}$, where $W_{89}^=$ denote the parameter associated with the connection between neuron 9 in layer $=-1$, and neuron 8 in layer $=$. We also let $<^=$ denote the number of neurons in layer $=$ and let $\#$ be the number of layers in our network. Therefore, for layer $= \in \{2, \cdots, \#\}$, we write $u^{C,=} = [D_1^{C,=}, \cdots, D_{<^=}^{C,=}]$ and $z^{C,=} = [I_1^{C,=}, \cdots, I_{<^=}^{C,=}]$ to denote the membrane and output vector of neurons in layer $=$ at timestep C. For $= = 1$, we will use $z^{C,1} = x^C$ to denote the input vector. Thus, the expression of an SNN is given by:

$$u^{C,=} = g u^{C-1,=} (1 - z^{C-1,=}) + W^= z^{C,=-1}, \quad (3)$$

$$z^{C,=} = \Theta(u^{C,=} - \vartheta_{th}) \quad (4)$$

From Equations 3-4, the spike signals not only propagate through the layer-by-layer spatial domain, but also a ect the neuronal states through the temporal domain. Therefore, it considers both the spatial and temporal directions during the error backpropagation, i.e., spatio-temporal backpropagation (STBP) [25, 26], which signi cantly improves the network accuracy. During backpropagation, because the activity function $\Theta(\cdot)$ is non-di erentiable, it is common to use the rectangular function to approximate the corresponding derivative.

Given the expressions above, we can easily solve a standard SNN classi cation problem by training a classi er $5 : R^{\%} \to \{1, \cdots, \#\}$ on a given dataset $\{x^1, y^1\}, \cdots, \{x^\sim, y^\sim\}$ that contains $\sim$ training samples, of which each instance $x^\wp \in R^{\%}$ has an observed label $y^\wp \in \{1, \cdots, \#\}$. $\%$ is the number of input entries and $)$ denotes the length of spike train. To train the SNN, we de ne the following loss function $!$ for a single training example (x, y):

$$! = \frac{1}{2} \Big(y - \frac{1}{)} \sum_C M z^{C,\#} \Big)^2 \quad (5)$$

where $z^{C,\#}$ denotes the voting vector of the last layer $\#$ at time step C, M denotes a constant voting vector connecting neurons in the output layer to a speci c class. Thus, we can use STBP to propagate the gradients $\frac{m!}{m I_8^{C,=}}$ from the $(=+1)$-th layer and $\frac{m!}{m u_8^{C+1,=}}$ from time step $C+1$ as follows:

$$\frac{m!}{m I_8^{C,=}} = \sum_{9=1}^{<^{=+1}} \frac{m!}{m u_9^{C,=+1}} \frac{m u_9^{C,=+1}}{m I_8^{C,=}} + \frac{m!}{m u_8^{C+1,=}} \frac{m u_8^{C+1,=}}{m I_8^{C,=}} \quad (6)$$

$$\frac{m!}{m D_8^{C,=}} = \frac{m!}{m I_8^{C,=}} \frac{m I_8^{C,=}}{m D_8^{C,=}} + \frac{m!}{m u_8^{C+1,=}} \frac{m u_8^{C+1,=}}{m D_8^{C,=}} \quad (7)$$

4 METHODOLOGY

In this work, we propose a sparsi cation procedure for deep SNNs that accelerates both training and inference while improving the their generalization capabilities through regularization.

Figure 3: Difference between traditional and sparse SNNs.

4.1 Sparsity regularization and optimization

To build a sparse structure, we consider a re-parametrization of $W^{(l)}_{ij}$ inspired by [13]:

$$W^{(l)} = \hat{W}^{(l)} \cdot b^{(l)}, \quad b^{(l)}_{ij} \in \{0, 1\}, \quad \hat{W}^{(l)}_{ij} < 0 \tag{8}$$

where the $b^{(l)}_{ij}$ correspond to binary gates that denote whether the corresponding parameter $\hat{W}^{(l)}_{ij}$ is utilized or not utilized. $\hat{W}^{(l)}$ and $b^{(l)}$ is also independent of time t. To simplify the later derivations, we reformulate the minimization of Equation 5 as $\mathcal{L} = \mathcal{L}(y, \hat{W} \cdot b)$.

By letting $q(b^{(l)}_{ij} \mid \theta^{(l)}_{ij}) = \text{Bern}(\theta^{(l)}_{ij})$ be a Bernoulli distribution over each gate $b^{(l)}_{ij}$ we reconsider a sparse network structure as a regularized minimization procedure with a regularization on the number of parameters being used, on average, as follows:

$$\mathcal{L} = \mathcal{L}_D + \alpha \mathcal{L}_C \tag{9}$$

$$\mathcal{L}_D = \mathbb{E}_{q(b)}\left[\mathcal{L}(y, \hat{W} \cdot b)\right] \tag{10}$$

$$\mathcal{L}_C = \sum_{l=1}^{L} \|\theta^{(l)}\|_1 \tag{11}$$

where $\mathcal{L}_D$ denotes the expectation of loss with respect to the Bernoulli distribution of b. Meanwhile, $\mathcal{L}_C$ corresponds to the complexity loss that measures the sparsity of the model. Due to the positive nature of each $\theta^{(l)}_{ij}$ this term also corresponds to the expectation of the amount of gates being on. Based on [16], the objective described in Equation 9 is a close surrogate to a variational bound involving a spike and slab distribution over the parameters and a fixed coding cost for the parameters when the gates are active. However, the first term in Equation 9 is problematic for θ due to the discrete nature of b, which does not allow for efficient gradient-based optimization. The unbiased gradient estimator in [24] could be employed, however, it suffers from high variance. The straight-through estimator in [3] can also be used in this problem, but it provides biased gradients as it ignores the Heaviside function during gradient evaluation.

In this paper, inspired by [13], we find a simple alternative way to smooth the objective function such that we allow for efficient gradient-based optimization of Equation 9. Let $s^{(l)}_{ij}$ be a continuous random variable with a distribution $q(s^{(l)}_{ij})$ that has parameters $\theta^{(l)}_{ij}$. We can now let each gate be given by a hard-sigmoid rectification of $s^{(l)}_{ij}$ as follows:

$$s^{(l)}_{ij} = q(s^{(l)}_{ij} \mid \theta^{(l)}_{ij}), \quad b^{(l)}_{ij} = \min(1, \max(0, s^{(l)}_{ij})) \tag{12}$$

This allows $b^{(l)}_{ij}$ to be exactly zero. Due to the i.i.d assumption of each $s^{(l)}_{ij}$ we can thus smooth the binary Bernoulli gates by replacing each $b^{(l)}_{ij}$ appearing in the first term of Equation 9 with $s^{(l)}_{ij}$ and the second term with the probability of the variable $s^{(l)}_{ij}$ being positive:

$$\mathcal{L}_D = \mathbb{E}_{q(s)}\left[\mathcal{L}(y, \hat{W} \cdot s)\right] \tag{13}$$

$$\mathcal{L}_C = \sum_{l=1}^{L} \sum_{ij} Q(s^{(l)}_{ij} > 0 \mid \theta^{(l)}_{ij}) \tag{14}$$

Here we similarly have a cost that explicitly penalizes the probability of a gate being different from zero, thus Equations 13-14 act as a close surrogate to the original loss function in Equation 10-11. By following the reparameterization trick [11], we can describe the expression in Equation 13 as an expectation over a parameter-free noise distribution $p(\epsilon)$ and a deterministic and differentiable transformation $g(\theta, \epsilon)$ of the parameter θ and ϵ. This allows us to make the following Monte Carlo approximation to the intractable expectation over the noise distribution:

$$\mathcal{L}_D = \frac{1}{M} \sum_{m=1}^{M} \left[\mathcal{L}(y, \hat{W} \cdot s^{(m)})\right] \tag{15}$$

$$s^{(m)} = \min(1, \max(0, g(\theta, \epsilon^{(m)}))), \quad \epsilon^{(m)} \sim p(\epsilon) \tag{16}$$

Next we provide more details about $g(\theta, \epsilon)$ in Equations 16.

4.2 The hard concrete distribution

The framework above enables us to employ efficient stochastic gradient-based optimization, while still allowing for exact zeros of the parameters. For the differentiable transformation $g(\theta, \epsilon)$, we follow [14]: assume that we have a binary concrete random variable B distributed in the interval (0, 1). The parameters of this distribution include $\log U$ and V, where $\log U$ denotes the location and V is referred to as the temperature.

Temperature V controls the degree of approximation. With $V = 0$, we recover the original Bernoulli distribution, whereas with $0 < V < 1$ we obtain a probability density that concentrates its mass near 0 and 1. Therefore the hard concrete distribution can inherit statistical properties very similar to that of the Bernoulli distribution. We then stretch B to the interval (γ, ζ) with $\gamma < 0$ and $\zeta > 1$. Following [14], we fix $\gamma = -0.1$, $\zeta = 1.1$ and all $V = \frac{2}{3}$ throughout this paper. Then we sample θ based on the expressions as follows:

$$B = \text{sigmoid}((\log U + \log \alpha) / V) \tag{17}$$

$$\bar{B} = B(\zeta - \gamma) + \gamma \tag{18}$$

$$\theta = \min(1, \max(0, \bar{B})) \tag{19}$$

Thus, the complexity loss $\mathcal{L}_C$ of the objective function in Equation 14 under the hard concrete distribution can be calculated as:

$$\mathcal{L}_C = \sum_{ij} \text{sigmoid}(\log \alpha^{(l)}_{ij} - V \log \frac{-\gamma}{\zeta}) \tag{20}$$

Algorithm 1 Training code for sparse SNN

Require: : i: Network inputs $\{-^{C}g_{C}$; ii: class label ; iii: parameters and states of convolutional layers $\{W^{=}\bullet b^{\cdot}\bullet u^{0\bullet=}\bullet o^{0\bullet=}g_{=1}^{\#_1}{}^{1}o$; iv: full-connected layers $\{(W^{=}\bullet b^{=}\bullet u^{0\bullet=}\bullet o^{0\bullet=}g_{=1}^{\#_2}{}^{1})$; v: simulation window T; vi: the parameters of the hard-concrete distribution $(\log U^{=}\bullet V\bullet W)$; vii: the parameters of iterative LIF $\{(:\ g\bullet X\bullet \dot{+}\ C\ )$

Ensure: : Update network parameters

Forward (inference) :

1: **for all** $C = 1$ to $)$ **do**
2: $b^{=}$ Generate $\{\log U^{=}\bullet V\bullet W^{\bullet}$ //Eq. (17)
3: $o^{C;1}$ EncodingLayer $\{-^{C}o$
4: **for all** $;$ = 2 to $\#_1$ 1 **do**
5: $^{1}u^{C\bullet=}\bullet o^{C\bullet=}$ StateUpdate $\{W^{=}{}^{1}\bullet b^{1}{}^{1}\bullet u^{C\ 1\bullet=}\bullet o^{C\ 1\bullet=}\bullet o^{C\bullet=1}\bullet x^{C\bullet=1}o$ //Eq. (21,22)
6: **end for**
7: **end for**

Loss:

! ComputeLoss $\{Y\bullet o^{C\bullet\#_2}\bullet \log U^{o}$ //Eq. (9)

Backward :

1: Gradient Initialization: $\frac{m!}{mo^{C;1\bullet}} = 0$
2: **for all** $C =)$ to 1 **do**
3: $\frac{m!}{mo^{C\bullet\#_2}}$ LossGradient $\{\dot{+}\bullet \frac{m!}{mo^{C;1\bullet\#_2}}o$ //Eq. (6,7,9)
4: **for all** $;$ = $\#_2$ 1 to 1 **do**
5: $^{1}\frac{m!}{mo^{C\bullet=}}\bullet\frac{m!}{mu^{C\bullet=}}\bullet\frac{m!}{mW^{=}}\bullet\frac{m!}{mU^{=}}o$ BackwardGradient $^{1}\frac{m!}{mo^{C\bullet=1}}\bullet\frac{m!}{mo^{C;1\bullet=}}\bullet W^{=}\bullet\log U^{=}o$ //Eq. (6,7,9)
6: **end for**
7: **for all** $;$ = $\#_1$ to 2 **do**
8: $^{1}\frac{m!}{mo^{C\bullet=}}\bullet\frac{m!}{mu^{C\bullet=}}\bullet\frac{m!}{mW^{=\ 1}}\bullet\frac{m!}{mU^{=\ 1}}o$ BackwardGradient $^{1}\frac{m!}{mo^{C\bullet=1}}\bullet\frac{m!}{mo^{C;1\bullet=}}\bullet W^{=\ 1}\bullet\log U^{=\ 1}o$ //Eq. (6,7,9)
9: **end for**
10: **end for**

Given these derivations, we can easily obtain the corresponding iterative state update equations and gradients for sparse deep SNNs.

$$u_{8}^{C;1\bullet=,1} = :g u_{8}^{C\bullet=;1}\ 1\ o_{8}^{C\bullet=;1}\ ,\ \tilde{O}^{o}_{9} W_{89}^{=} b_{89}^{=} o_{9}^{C;1\bullet=} \quad (21)$$

$$o_{8}^{C;1\bullet=,1} = K\ u_{8}^{C;1\bullet=,1}\ +_{\text{th}} \quad (22)$$

$$b_{89}^{=} = \min\ 1\bullet\max^{1}0\bullet s_{89}^{=}o\ \bullet \quad (23)$$

$$s_{89}^{=} = s_{89}^{=}{}^{1}e\ \ W^{\bullet}, W\bullet \quad (24)$$

$$s_{89}^{=} = f\ \ \log D\ \ \log^{1}1\ \ D^{o}, \log U_{89}^{=}\bullet V \quad (25)$$

We also summarize the overall training process of our proposed sparse SNNs as pseudo-code in Algorithm 1.

5 EMPIRICAL STUDY

To comprehensively validate the e ectiveness of our proposed method, we conduct experiments to answer two questions: First, we are interested in computational improvement with very negligible degradation in accuracy. Our work in this paper thus aims to improve the state-of-the-art SNN in this regard. We choose the Spiking CNN (SCNN) [26] as the basic model to which we apply our proposed sparsi cation procedure on this model and name it sparse SCNN. We then compare the e ciency and accuracy of our Sparse SCNN with SCNN, M-SNN [5], and stochastic SNN [1] on various classi cation tasks. To better compare our work with them, we follow the same experimental setting as in [26], including the same experimental datasets and the same network structure. Second, we want to explore the generalizability of our proposed model, especially for high dimensional data with very few training samples. We thus test on small training subsets of MNIST and N-MNIST. We validate our sparse deep SNN framework by using the state-of-the-art fully connected and convolutional architectures for deep SNNs [26] on these datasets. To combat randomness in the experiment system, we run all experiments 10 times and report the average results, except when otherwise stated.

5.1 Datasets

We evaluate our sparse SNN models and baselines on various datasets. Using the same datasets as in [26], we test on both static (non-spiking) as well as dynamic (neuromorphic) data.

5.1.1 Static Datasets. MNIST is a popular dataset comprised of a training set with 60 000 samples and a testing set with 10 000 samples of hand-written digits 0 9. CIFAR-10 is an established computer-vision dataset used for object recognition. It consists of 60 000 32 32 color images containing one of 10 object classes, with 6 000 images per class. Since our method and baselines are spike based learning algorithm, the static images should be converted to spike trains. To this end, we use the Bernoulli sampling conversion from original pixel intensity to the spike trains in this paper. Each normalized pixel is converted to a spike event 1) or no spike event 0) at each time step by using an independent and identically distributed Bernoulli sampling. The probability of generating a spike event is proportional to the normalized value of the entry. Thus, given a certain time window , the spike events form a spike train. During training, we set) to 12 and 30 ms in MNIST and CIFAR-10, respectively.

5.1.2 Dynamic Datasets. Compared to the static datasets, dynamic datasets contain richer temporal features and are therefore more suitable for evaluating SNNs since SNNs can take advantage of the added information. We use the N-MNIST [19] and DVS-Gesture [2] datasets to evaluate the capability of our method on dynamic datasets. The N-MNIST dataset [19] consists of MNIST images converted into a spiking dataset using a Dynamic Vision Sensor (DVS) moving on a pan-tilt unit. Each dataset sample is 300 ms long, with a shape of 34 34 pixels, containing both on and o spikes. The dataset is split into training and test sets following the original split in MNIST of 60 000 training samples and 10 000 testing samples.

The DVS-Gesture dataset [2] contains 1 342 instances of a set of 11 hand and arm gestures, grouped into 122 trials and collected from 29 subjects under 3 di erent lighting conditions. During each trial, one subject stood against a stationary background and performed all 11 gestures sequentially under the same lighting conditions.

[1] https://www.garrickorchard.com/datasets/n-mnist
[2] https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8

Table 1: Fixed parameter values for the various experiments.

Parameter	Description	Chosen Value (MNIST/CIFAR10/N-MNIST/DVS-Gesture)
)	Time window	30ms,12ms,300ms,1450ms
: g	Decay factor	0.1ms,0.3ms,0.2ms,0.2ms
X	Derivative approximation parameter	1.0,0.5,0.5,0.5
$+_{th}$	Threshold	0.5
V	Temperature of hard-concrete distribution	2/3
W • e	Other parameters of hard-concrete distribution	-0.1, 1.1
_	The weight factor of sparse regularization	0.001

Figure 4: Comparison with SCNN[25], M-SNN[5] and stochastic SNN[1] on MNIST. We show the number of million oating point operations (MFLOPs) after training for each model. These were computed by assuming one FLOP for multiplication and another FLOP for addition.

These gestures are recorded using a DVS128 camera, which is a 28 28-pixel Dynamic Vision Sensor. The problem is to identify the correct action label associated with each action sequence video.

5.2 Network structure

Throughout this section, we use the following notations to describe the deep SNN architecture. Layers are separated by and spatial dimensions are separated by . A convolution layer is represented by and a pooling layer is represented by %. For example, 28 28 15 5 %2 10 represents a 4-layer spiking CNN with 28 28 input, followed by 15 convolution lters that are 1 5 5, followed by 2 2 pooling layer and nally a dense layer connected to 10 output neurons. Table 2 provides the network structures for experiments. We use the exact same network architecture for our model and baselines for a fair comparison.

Table 2: Network structures used for experiments.

Static Dataset	
MNIST	28 28 15 5 %2 40 5 %2 300 10
CIFAR10	34 34 2 32 3 %2 64 3 %2 256 10
Dynamic Dataset	
N-MNIST	34 34 2 16 5 %2 32 3 %2 64 3 10
DVS-Gesture	128 128 2 %4 16 5 %2 32 3 %2 512 11

5.3 Initialization

In our proposed model, some parameters, such as the model weights and the locations of the hard-concrete distribution, need to be learned while others need to be xed throughout the optimization. We now discuss our choice for initializing these parameters, which includes the weights, the thresholds and the decay factor for each neuron, the weighting factor for the sparse regularization, and the parameters of the hard-concrete distribution. We divide these parameters into two sets to consider.

First, to better mimic the neural dynamics, we need to control the relative magnitude between the weights and thresholds to avoid too much spiking, which reduces neuronal selectivity. In practice, and as a simpli cation, we x the threshold value as a constant for each neuron and only adjust the weights that is responsible for controlling/balancing activity. We initialize all the weight parameters by sampling from the standard uniform distribution followed by normalization.

Second, while sparsifying the network, we follow [14] and set W= 0 1 e= 1 1 V= $\frac{2}{3}$ for the concrete distributions. Meanwhile, we initialize the locations logU by sampling from a normal distribution

Figure 5: Comparison with SCNN[25], M-SNN[5] and stochastic SNN[1] on CIFAR-10.

Figure 6: Comparison with SCNN[25], M-SNN[5] and stochastic SNN[1] on N-MNIST.

with a standard deviation of 0.01 and a mean of 1. In practice, we use a single sample of the gate for each mini-batch of the dataset during the training, even though this can lead to larger variance in the gradients. This way, we show that we can obtain the speedups in optimization with a practical implementation without incurring a significant loss in classification accuracy. A summary of the values of the fixed parameters used is shown in Table 1.

5.4 Evaluation metrics

To evaluate classification performance, we use the standard Accuracy metric. To evaluate the computational efficiency, we count the floating point operations (FLOPs) to measure the potential speedup. FLOPs are computed by assuming one flop for multiplication and one flop for addition.

5.5 Experiment results

In this section, we discuss the experimental results pertaining to each of the two previously-raised research questions separately.

5.5.1 Potential speedup. The tables shown in Figures 4 and 5 compare our proposed sparse deep SNNs with a traditional SNN, M-SNN, and stochastic SNN on the static MNIST and CIFAR-10 datasets, respectively. Even without a complex architecture, the proposed deep SNNs and their competitors still perform well on these datasets. We find that there is only a slight difference in accuracy between our sparse deep SNNs and their competitors, i.e. (only between 0.05% and 0.1%). This is a negligible difference. However, as we can observe, there is a significant improvement in the FLOP count between our sparse deep SNNs and the competitors. On CIFAR-10, our sparse network and M-SNN incurs only half the computational cost compared to traditional SNN. On MNIST, this ratio is further reduced to less than 25%, which allows for a potentially significant speedup in inference phase.

Table 3: Comparison on small datasets. Best accuracy highlighted.

Dataset	Method	Accuracy
MNIST	SCNN	69%
	Sparse SCNN (Ours)	92%
N-MNIST	SCNN	95%
	Sparse SCNN (Ours)	97%

Figure 7: Comparison with SCNN[25], M-SNN[5] and stochastic SNN[1] on DVS-Gesture.

The results for the neuromorphic datasets are shown in Figures 6 and 7. We find that both M-SNN and stochastic SNN, which perform well on static datasets, have a significant degradation in accuracy. This demonstrates that the works proposed in [1, 5] are not as suitable for neuromorphic datasets. Meanwhile, by using a sparse network structure, our proposed model incur a slight degradation in accuracy (i.e., decrease between 0.05% and 0.4%) but sparsity can provide a significant speedup of nearly 5 times. Our experimental results show that sparsifying deep SNNs using our proposed framework can greatly speed up training and inference while only incurring a minimal and negligible loss in classification performance. In summary, our proposed model achieves better computational efficiency than previous works when tested on both neuromorphic as well as static datasets and achieves very negligible degradation in accuracy.

5.5.2 Better generalization. To evaluate the ability of the model to generalize, we first compare the performance of our proposed method to SCNN on MNIST as the size of the training set is varied. We continuously reduce the training set of MNIST and test on a test set of the same size.

As shown in Figure 8, although both methods achieve very competitive accuracy when the whole training set is used, our sparse deep SNN demonstrates much higher robustness when training set size is decreased. In particular, we observe that the performance of the non-sparse model drops sharply when the training set size is reduced to below 3,000 while the sparse deep SNN's performance remains fairly steady. We conclude that when there are not enough training samples, deep SNNs will easily overfit and even memorize random patterns in the training set. This overfitting can lead to poor generalization. In contrast, by using sparse architecture in deep SNNs, the model shows better generalization even when training samples are limited.

We summarize the results of these experiments, run on MNIST and N-MNIST, in Table 3. During training, we limit the percentage of available training samples to only 1.67% (i.e., only 1,000 samples). As can be observed from the results, all the sparse deep SNNs demonstrate higher accuracy than that of their competitors. This demonstrates that by inducing model sparsity in the architecture, deep SNNs can achieve better generalization in practice.

Figure 8: Generalization test on MNIST dataset. The Y-axis denotes the accuracy of each model on the test set (10,000 samples). The X-axis denotes the size of the training set.

5.6 Impact of the weight factor of sparse regularization

In this paper, we achieve sparsity in the network structure of deep SNNs via sparsity regularization, which balances the accuracy with the percent of non-zero weights. Now we quantitatively analyze the impact of this sparsity regularization. We implement a sparse spiking CNN on MNIST, keeping the model configurations the same as our previous experiment on MNIST (28×28-15C5-P2-40C5-P2-300-10).

In Figure 9, the left side shows the level of sparsity at each layer when $\lambda = 0.001$. We use subgraphs of different widths to correspond to the number of weights of each layer in the network, while using the height of blue shaded area to correspond to the sparsity of each layer. The right side shows the overall level of sparsity under different values of λ. The X-axis denotes the value of the sparsity regularization term while the Y-axis denotes the percent of non-zero weights.

The results in Figure 9 show that sparse regularization has a different influence on convolutional layers and fully connected layers. One reason why the the fully connected layers are sparser than the convolutional layers may be due to the difference in nature