




Efficient Processing of Spiking Neural Networks via Task Specialization

Muath Abu Lebdeh , Kasim Sinan Yildirim , *Member, IEEE*, and Davide Brunelli , *Senior Member, IEEE*

Abstract—Spiking neural networks (SNNs) are considered as a candidate for efficient deep learning systems: these networks communicate with 0 or 1 spikes and their computations do not require the multiply operation. On the other hand, SNNs still have large memory overhead and poor utilization of the memory hierarchy; powerful SNN has large memory requirements and requires multiple inference steps with dynamic memory patterns. This paper proposes performing the image classification task as collaborative tasks of specialized SNNs. This specialization allows us to significantly reduce the number of memory operations and improve the utilization of memory hierarchy. Our results show that the proposed approach improves the energy and latency of SNNs inference by more than 10x. In addition, our work shows that designing narrow (and deep) SNNs is *computationally more efficient* than designing wide (and shallow) SNNs.

Index Terms—Spiking neural networks, low energy and latency computing, memory aware SNNs, specialized SNNs.

I. INTRODUCTION

NEUROMORPHIC computing offers a brain-inspired approach to enable intelligence by reducing the power and energy requirements of computing platforms [1]. Neuromorphic systems use the spiking neural network (SNN) as a computational model, in which each neuron uses the heaviside step function as an activation function to generate spikes for communication with other neurons. This pushes the activations to be sparser (e.g., compared to the ReLU activation function), leading to less effectual operations, and hence to better utilization of event-driven computing resources. Moreover, SNNs rely on a dynamic neural model that makes them capable of capturing spatio-temporal patterns at the neuron level without any architectural recurrent connections [2]. Therefore, this relatively complex neural model allows feed-forward SNN architectures to be deployed in a wide range of applications [2], [3], [4]

On the other hand, due to their dynamic properties, every neuron in the SNN requires a dynamic state variable (the membrane potential) to be maintained in memory during the whole

inference. This is an additional memory overhead compared to feedforward ANNs. In feedforward ANNs, all intermediate results produced by any layer can be completely evicted from memory after producing the output of the next layer, while in a feedforward SNN, the membrane potential variable of every neuron in the network needs to be maintained in the memory during the whole inference time. This additional overhead results in two challenging problems when deploying SNNs in resource-constrained computing systems:

- 1) *Significant reduction in efficiency*: With a limited working memory (e.g., the main memory in a conventional computing system or the on-chip memory of a neuromorphic system), the membrane potential variables of a large scale SNN need to be moved to a denser storage device (e.g., FLASH memory). This results in more frequent accesses to the storage device during runtime, and hence resulting in inefficient read and write operations. This increases the overall energy consumption and latency of the SNN inference, which squanders the other efficiency benefits of SNNs such as the sparse computation and the replacement of the multiply-and-accumulate operations with accumulate operations.
- 2) *Increasing the cost of the computing system*: As we need to store all membrane potential variables in some form of memory (either working memory or a storage device), we increase the memory requirements of the computing system. Therefore, the overall cost of the computing system increases. This is especially obvious when we desire to increase the size of the working memory. For example, intel Loihi 2 neuromorphic chip has 128 asynchronous neural cores, each including up to 128 KB of synaptic memory and up to 8192 neurons. This limited on-chip memory requires partitioning of large scale SNNs into smaller computations to fit the working memory of the chip [5]. This requires either increasing the on-chip neural capacity, using multiple neuromorphic chips, or using an external memory and sequentially executing the partitioned computations. All these solutions increase the overall cost, which is undesirable for applications with severe constraints on computational resources.

Contribution: In this paper, we propose utilizing the idea of task specialization to design narrow SNNs, and hence reduce the memory overhead of SNNs. This reduction in memory overhead includes reducing the number of memory operations and increasing the utilization of memory hierarchy. The proposed approach performs the classification task as a collaborative effort

Manuscript received 29 May 2023; revised 9 December 2023; accepted 29 January 2024. This work was supported in part by the GEMINI “Green Machine Learning for the IoT” national research project, funded by the MUR through the PRIN 2022 program under Grant Prot. n. 20223M4HZ4. (Corresponding author: Muath Abu Lebdeh.)

Muath Abu Lebdeh and Kasim Sinan Yildirim are with the Department of Information Engineering and Computer Science, University of Trento, 38122 Trento, Italy (e-mail: muath.abulebdeh@unitn.it).

Davide Brunelli is with the Department of Industrial Engineering, University of Trento, 38122 Trento, Italy.

Recommended for acceptance by M. Zhang.

Digital Object Identifier 10.1109/TETCI.2024.3370028

of multiple specialized and narrow SNNs rather than being performed by a single and fairly large SNN. We show that training each specialized SNN on a relatively unique subset of the dataset leads to narrower layers, and in turn, decreases the spatial dependencies between consecutive layers in SNNs. This reduction in spatial dependencies reduces the number of needed memory and computing operations. In addition, the decrease in spatial dependencies allows the specialized SNNs to fit better into more local memories, which reduces the movement of data in different levels of the memory hierarchy. Therefore, our proposed approach reduces the overall inference energy and latency.

Briefly, the contribution of this paper is summarized below:

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

The rest of this paper is organized as follows. Section II presents the related work to our goal and approach. Section III gives preliminary background on SNNs, Section IV explains the main aspects of our approach. Section V presents the evaluation methodology and the results. Section VI discusses some limitations and future work. Finally, section VII concludes the paper.

II. RELATED WORK

Our work aims at using specialized SNNs for the goal of reducing the memory overhead of SNNs; namely, reducing the number of memory operations and increasing the utilization of memory hierarchy. There are several related works in the literature that either (1) share a similar purpose of reduce SNNs' computational overhead in general and the memory overhead in specific while using different and orthogonal approaches, (2) share a similar approach of using an ensemble of weak SNN classifiers but not directly to improve the processing efficiency of the SNNs, or (3) share a similar approach of using specialized networks for reducing the ANNs computations but not for reducing the memory overhead of the SNNs. This section discusses these three types of related work.

Reducing the memory overhead of SNNs: Our work has a very similar goal to the work in [6]. This work aims at reducing the number of membrane potentials at any time step, which automatically decreases the memory requirements of the SNNs, the number of memory operations as well as the utilization of the memory hierarchy. Although this work has the same goal of our work, we take a different direction to reduce the memory overhead. We reduce the memory overhead via specialized ensembles of SNNs, which does not overlap with the approach in [6]. In other words, these approaches are orthogonal and can

be integrated; e.g., the approach in [6] can be applied to the specialized SNNs and further reduce their memory overhead.

SNN ensembles: The work in the literature that used ensembles of SNNs have different goals than decreasing the memory overhead of SNNs. In [7], SNN ensembles were used to facilitate the training of SNNs, and decrease the overall number of parameters. Even though this work improves the processing efficiency of SNNs, it still does not directly address the high memory overhead in SNNs. In addition, the datasets tested in this approach are relatively simple. In [8], two unimodal SNN ensembles were used to enable multimodalities in SNNs with STDP training. The focus of this paper is to enable a new application in SNNs rather than improving the processing efficiency of SNNs. In [9], an ensemble of weak SNN classifiers were used to reduce the overhead of extracting SNN architecture using evolutionary algorithms. This work does not target improving the processing efficiency of SNNs; in fact, they use an ensemble of 50 SNNs to implement a toy application (UCI handwritten digits). In [10], HybridSNN was proposed as an adaptable architecture that is made of a homogeneous ensemble of SNNs, ranging from single-layer SNNs to convolutional SNNs. The goal of this approach is to stabilize and facilitate the training of SNN (in a modular and biologically plausible way). Although this approach also can improve the processing efficiency of SNNs, it does not directly target reducing the memory overhead of SNNs. The HybridSNN can learn architectural patterns with wide layers (layers with large number input or output features) and hence increased memory overhead.

Sparsely-gated mixture-of-experts (SG-MoE): Our work has a similar (but simpler) approach to the work of SG-MoEs [11], [12], [13] while targeting a different goal. These papers aim at skipping computations and hence increase the processing efficiency of ANNs. Although this also can be valid for our approach, we aim at utilizing the notion of expertise to systematically decrease the width of the specialized SNN models, and hence reduce the memory operations and increase the utilization of memory hierarchy. Note that with some modifications to the training in our approach, we can also utilize the same approach for a novel computation skipping in SNNs. In addition, the work that deploys the SG-MoE in image classification applications relies on vision transformers [12], such that the input image is applied sequentially as patches. Each patch can be processed by a small set of models from the whole set of MoEs (e.g., some patches visually present the background, other patches present specific geometrical shapes, etc). In our work, we implement the specialty for the SNNs without patching the input image, such that we consider the whole input image as a patch that can be processed by a specialized SNN.

Other general approaches to improve the processing efficiency of SNNs: A group of work re-used the network compression techniques via pruning and quantization for SNNs [14], [15], [16]. These techniques reduce the model size by reducing the neurons count (e.g., eliminating neurons with low firing rates), reducing the synaptic connectivity, or reducing the bit size of the network's parameters. These compression techniques still do not push the energy and latency of an SNN to a fundamentally low level; for example, unstructured pruning is a

common technique used to remove redundancy, while it does not fully benefit from sparse tensor accelerators [17] which require structured input data (matrices and vectors). Therefore, compression techniques are mainly useful for improving the storage and memory requirements of the SNNs, which indirectly improve the energy and latency.

Another group solutions penalize the SNN during training to reduce the spikes count [18], [19]. These solutions allow the SNNs to make better utilization of sparse and asynchronous computing hardware. However, as our results show (in Section V; evaluation), reducing the spikes count alone does not necessarily reflect the efficiency of processing an SNN. The energy consumption and latency depend on both the spikes count per layer and the width of the next layer.

Other work targets reducing the number of time steps [20], [21], [22]. These techniques are especially useful for SNNs converted from ANNs. Reducing the number of time steps is equivalent to reducing the sequential sub-inferences required for complete SNN inference. As fewer sequential sub-inferences are required, this improves the overall energy and latency. On the other hand, although reducing the time steps improves the efficiency of a specific SNN, time steps alone is not a metric for latency as well as energy. For example, an SNN with low spiking activity and a high number of steps can be faster and more energy-efficient than an SNN with a low number of time steps but high spiking activity.

Another promising group of techniques utilizes special encodings to reduce the energy and latency of SNN [23], [24], [25], [26]. These encodings rely on using a sparse representation of the input data to reduce the overall spike count in the SNN. For example, in delay encoding, each input feature is represented by two spikes, where the amplitude of the feature is represented by the number of time steps between these two spikes. On the other hand, this low number of spikes in the input results in vanishing outputs in deep layers of SNNs, which makes training deep SNNs difficult [23]. Moreover, representing information using a low number of spikes makes the inference more prone to error [23], [24].

III. BACKGROUND

This section briefly presents a background on SNNs, encoding the input and the reading output in SNNs, and finally the training algorithms for SNNs.

A. Spiking Neural Networks (SNNs)

SNNs are similar to conventional ANNs except that they use spiking neurons. As shown in Fig 1, the spiking neuron receives a spike as an input ($S^{L-1}[t]$), which accumulates with time in an internal state variable called the membrane potential ($v^L[t]$). If the membrane potential exceeds a threshold value (v_{th}^L), the spiking neuron produces an output spike ($S^L[t]$). The state of membrane potential decays with time using a decaying function; in Fig. 1, the decaying function is a linear function with a decaying factor (α). One can think of the operation of the spiking neuron as an accumulator of dot product operations (correlations) between a stored feature vector (the set of weights

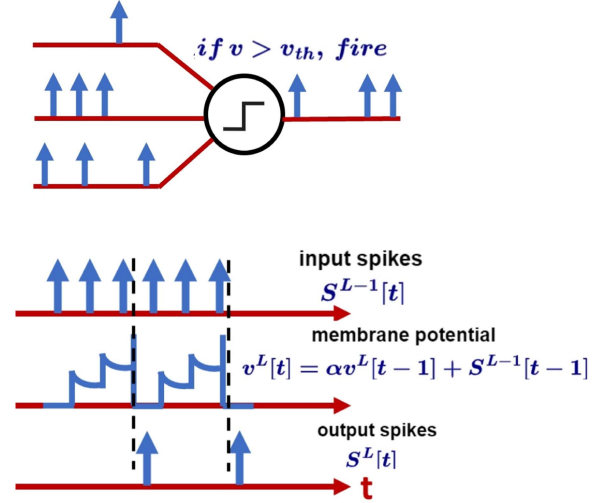


Fig. 1. Spiking neuron receives spikes as input and produces spikes as output. The input spikes are accumulated in the membrane potential ($v^L(t)$), and an output spike is generated if $v^L(t) > v_{th}$.

connected to the spiking neuron) and a set of boolean feature vectors applied sequentially (one vector is applied to the spiking neuron in every time step). In addition, the decaying factor plays a role in forming short- and long- term memories within the SNN [2]. These aspects allow the SNNs to learn spatio-temporal patterns at the neuron level [2].

B. Data Encodings in SNNs

SNNs are dynamic networks that are capable of processing input data in spatio-temporal manner. This allows for different ways of organizing input data and different ways to interpret the output data. This makes SNNs applicable to a wide range of applications.

In image classification application, an input image can be encoded in different ways such as rate encoding, delay encoding and direct encoding [27]. In rate encoding, each pixel value is encoded into multiple one-bit values (spikes). The frequency of these spike encodes the intensity of every pixel; i.e., higher value pixels are encoded by more spikes, each of which is applied in a single time step. Delay encoding encodes each pixel value into two spikes of specific temporal distance. For example, if the pixel value is higher, the distance between the two encoding spikes will be large. This type of encoding is energy efficient as it uses only one spike per pixel. Finally, direct encoding applies the whole input image as it is, while the first layer automatically encodes the pixels to spikes. We adopt the direct encoding in our approach as it produces higher accuracy in image classification applications [24].

The output of an SNN can be also interpreted differently. The most common way in the static image classification application is to dedicate one output spiking neuron per class. At the end of the inference operation, the output neuron that fires spikes the most corresponds to the result of the inference. We adopt this way of interpreting the output in our work. Another less common way to interpret the output is by assigning the output label to the

neuron that fires first (or last). This approach is usually used along with delay encoding at the input.

C. Training SNNs

Backpropagation through time (BPTT) is commonly used for the supervised training of SNNs. However, the training of SNNs using BPTT is more challenging than ANNs. The Heaviside activation function used by the spiking neurons vanishes the gradients during backpropagation. Therefore, surrogate gradient descent (SGD) replaces the derivative of the Heaviside function with a smoother function during the backward step [28], [29].

There are also other techniques that can be used to train the SNNs such as the biologically-inspired learning, i.e., Spike-timing-dependent plasticity (STDP) [30] or ANN-to-SNN conversion techniques [31], [32]. These techniques either work with shallow SNNs (such as the STDP learning) or produce SNNs with inefficient spiking activity (such as ANN-to-SNN conversion) [28]. Therefore, we adopt in this work the use of BPTT to train the SNNs.

IV. SPECIALIZED SPIKING NEURAL NETWORKS

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

A. Reducing the Width of the SNN Model

The problem of wide SNNs: Powerful SNNs that target image classification tasks are usually deep and *wide* [23]. Wide layers are capable of memorizing more features, which gives the backpropagation algorithm a more relaxed optimization space to find a solution. On the other hand, a wide layer has larger dependencies that result in more memory operations and increased data shuffling within the memory hierarchy.

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

Task specialization: On the other hand, narrowing the width of an SNN reduces the memorization capacity of the SNN, which makes the training more difficult. This challenge is alleviated by simplifying the classification task, which relaxes the memorization requirements. Therefore, we split the image classification

task into multiple simpler tasks, each of which is executed by a narrow specialized SNN. We refer to each specialized SNN as a *sub-SNN* which performs a sub-task. The next subsection presents more details the design of the specialized sub-tasks.

B. Designing the Specialized Tasks

We specialized every sub-SNN to detect one class in the image classification application, i.e., each sub-SNN is a binary classifier for one class. Image classification applications usually have classes that are mutually-exclusive, such that every class corresponds to one visual object. Based on this, there must be a level of independence between the visual features represented by every class, especially at deeper layers ([33] suggested that the mutual information between the output layer and a hidden layer increases with depth). Conventionally, a single SNN is used to classify all classes. This imposes every layer to learn the specific features of every class. During the inference, this imposes computing all features corresponding to all classes, while the input image belongs to one class. Instead of that, we use a specialized SNN (sub-SNN) for each class in the image classification problem. Therefore, each sub-SNN will learn the most relevant features from the images of the corresponding class. This allows us to remove all unnecessary computations for irrelevant features (the features for other classes) with all corresponding dependencies. Therefore, we reduce the overall memory overhead (less memory operations and better utilization of the memory hierarchy).

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

C. Training Via Heterogeneous Batching

As described in the previous subsection, each sub-SNN is trained by a specialized set of data. This specialized set of data contains two classes of samples: (1) the target class samples and (2) the samples corresponding to the other classes or the alien class samples. For example, if we have CIFAR10 dataset as the baseline dataset, one sub-SNN is specialized in detecting the airplane object. The corresponding specialized set of data contains: (1) The target class data which contains samples of the airplane object, while (2) the alien class data contains samples from other objects (i.e., automobile, bird, cat, etc). One way to organize the training batches in this specialized set of data is by sampling 50% of the samples from the target class and the other 50 from the alien class.

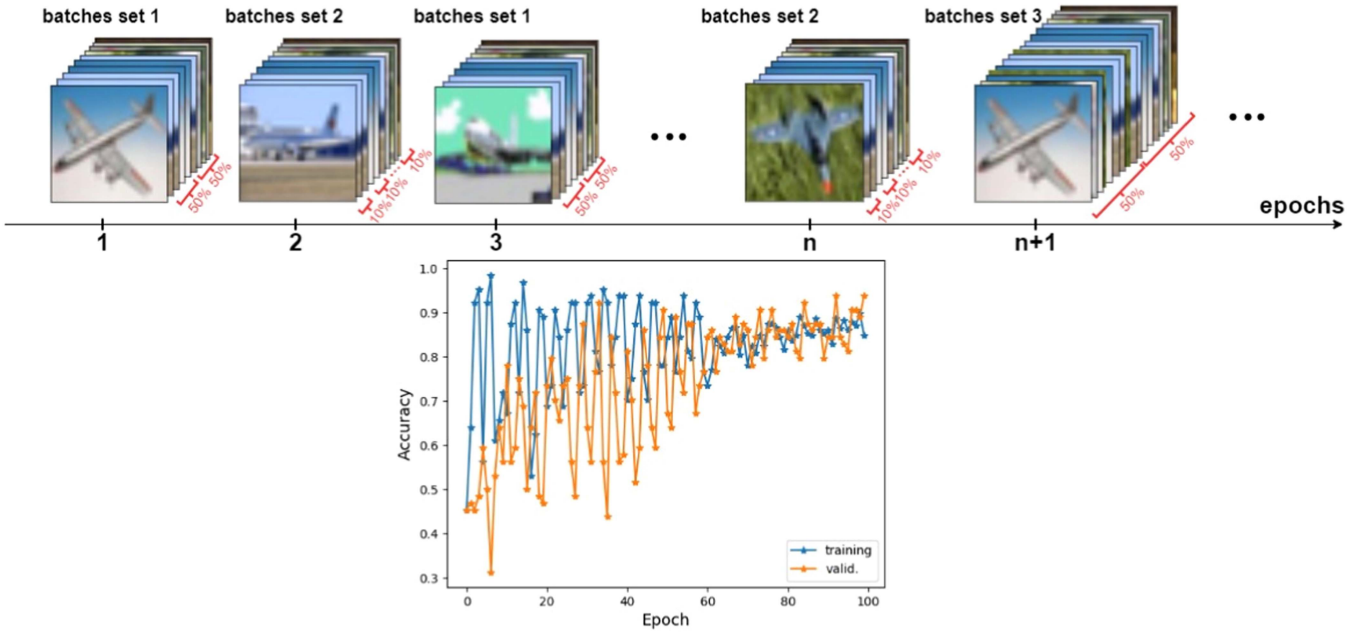


Fig. 2. Training each sub-SNN with heterogeneous sets of batches. Set 1 contains 50% of the samples from the target object and the other 50% from all alien objects. The samples in set 2 are sampled with equal percentages from all objects. Set 3 is the same as Set 1 except that this set contains larger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

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

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

The sum of weighted gradients: Another way is to weight and accumulate the gradients of more samples in the alien class. In this approach, every batch contains equally distributed samples from every object (e.g., 32 samples from all objects). After that the following steps are applied: (1) the gradient is computed for all samples (32×10 samples), (2) the gradients corresponding to each object are averaged (the result is 10 different gradients), (3) the gradients corresponding to the alien class are averaged (each averaged gradient corresponding to the alien class is weighted by $1/9$), (4) the final values are summed and the sub-SNN parameters are updated. Even though this approach produces high accuracy, it is time-consuming (for example, a conventional batch of 64 samples is scaled to 320 in this approach).

Heterogeneous batching: As shown in Fig. 2, we adopted a hybrid solution that is a composite of the aforementioned approaches. Instead of the sum of weighted gradients, we train using heterogeneous set of batches. The first set of batches is composed of 50% of the samples from the target class and other 50% from the alien class. The second set is composed of samples equally distributed among all classes. This set of batches has

more samples corresponding to the alien class. Therefore, the gradients of this batch is scaled by $1/c$ (c is the total number of objects in both classes). The training is performed by alternating between these two sets of batches at the epoch level rather than at the iteration level; i.e., the set of batches alternate at every epoch (e.g., set1 is applied at even epochs and set2 on odd epochs). Finally, as alternating between different batches still produces fluctuations in the learning curve, after some epochs, a new set of large batches is applied with 50%-50% distribution (similar to the first set of batches but with a larger batch size). These sets of large batches are applied once the loss reaches a desired value. With this training approach, the accuracy becomes similar to the approach of weighted sum of gradients but with faster training time.

D. Sub-SNN Structuring

A classification task of C classes is divided into C sub-classification tasks, each of which is executed by a sub-SNN. The baseline SNN architecture is split into C sub-SNNs where each sub-SNN is specialized in detecting only one class. Fig 3 shows the workflow of structuring each sub-SNN.

Initially, a given baseline SNN architecture is trained on the whole dataset. The purpose of training this baseline SNN architecture is to reuse the trained weights when creating the specialized sub-SNNs.

Each sub-SNN is initially structured by halving the width of every layer in the baseline SNN architecture (i.e., a scale factor of 2). This is done by removing half of the feature maps in every layer. To be able to benefit from the trained weights in the baseline SNN, halving the width is implemented by zeroing the weights (or convolution kernels) corresponding to the feature maps with the least activity. This ensures that the

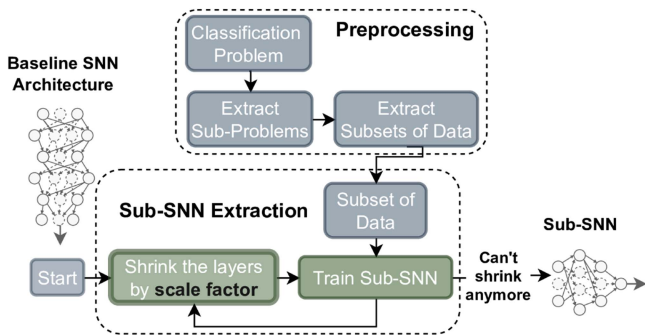


Fig. 3. Sub-SNN is structured by reducing the width of the baseline SNN, then training it on a subset of the dataset, then comparing the sub-SNN accuracy to the baseline SNN. This process is repeated until the scaled sub-SNN does not meet the accuracy of the baseline SNN.

most relevant features to the specialized task is maintained. If a desired accuracy is reached, the baseline SNN architecture is replaced with the new SNN architecture of halved width. After that, the same process is repeated until we reach an SNN architecture that cannot have narrower widths but maintains the baseline accuracy. In fact, we choose in our experiments a scale factor of 2 to speed up the training process. Other fine-tuned scaling techniques for the width can also be adopted.

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

E. Introducing Dependencies Between the Deepest Features Via Simple Aggregation Function

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

Fig. 4 shows the average spiking activity of the output neuron of every sub-SNN for objects from different classes. As seen in this figure, some sub-SNNs generate, with a high probability, false positives for some objects. This means that employing a trivial decision-making from the sub-SNNs, e.g., tracking the output of the sub-SNN that fires, does not result in sufficient accuracy.

Aggregating the results of the sub-SNNs: Even though some sub-SNN generates false positives with high probability, these false positives correspond to specific objects (not distributed for all objects). We utilized this consistency in false positives

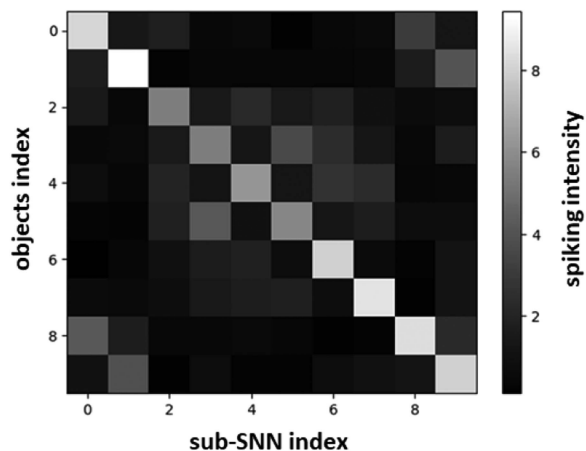


Fig. 4. This figure shows the spiking intensity of different sub-SNNs (in this example we have 10) when samples from different objects is applied (here we have 10 objects).

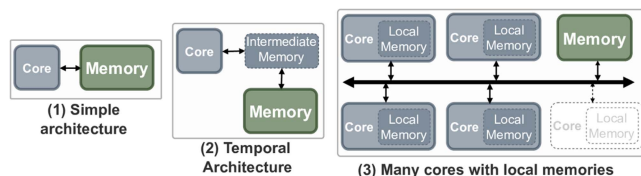


Fig. 5. Abstraction of different typical computing architectures.

to combine the deepest features (i.e., the output of every sub-SNN) using a simple *aggregation function*. This introduced dependencies at the output retains the overall accuracy of the baseline SNN, while significantly improving the computational efficiency due to the elimination of dependencies at low-level features. To not introduce a computational overhead, a simple *weighted average voting* aggregation function is used. For every final output (the output corresponding to each class), this function takes every output of the specialized sub-SNN, multiplies it by weight and adds it to the final output. This assigns a weighted dependency to every sub-SNN based on the target output class. The values of these weights are separately trained after training all sub-SNNs; i.e., the training dataset is reused along with the trained sub-SNNs to train these weights to produce the final classification results.

F. The Efficiency Potential of Using Sub-SNNs

The proposed approach has three potential ways to be utilized for the efficient processing of the SNNs. These potentials can be observed with the use of the computing architectures in Fig. 5. The simple architecture can utilize the reduced number of memory and computation operations, which reduces the overall latency and energy; ultra-low power microcontrollers (such as TI MSP430) are typical examples of a simple architecture. The temporal architecture can utilize the reduced reuse distance of the sub-SNNs due to the narrower layer's width. This allows part of the memory operations to access data in the efficient intermediate memory. Examples of this architecture are the

Algorithm 1: Executing a FC layer L in SNN.

```

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

```

computing systems with on-chip and off-chip memories. Finally, architectures with multiple processing elements can utilize the new level of parallelism the sub-SNNs introduce; i.e., the computations of every sub-SNN are completely independent of other sub-SNNs, allowing each sub-SNN to run on a separate processing element. Examples of spatial architectures include multi-core processors and neuromorphic chips.

V. EVALUATION

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

A. Evaluation Methodology

Experimental setup: We used `snnTorch` to integrate the SNNs under PyTorch. We evaluated our proposed approach for image classification problems using MNIST and CIFAR10 datasets and also for speech commands classification using Google speech commands V2 (we used 13 classes picked). We evaluated our work on 4 baseline architectures: (1) MLP trained on MNIST, (2) VGG9 trained on CIFAR10, (3) VGG9 with skip connections (called here `resnet`) on CIFAR10, (4) Google speech commands (called here `Key Word Spotting - KWS`) trained on 3 fully connected layers (3FCLs). In every architecture, all neurons are homogeneous, i.e., they have the same threshold and decaying factor. We used our sub-SNN structuring approach to structure the VGG9, RESNET and 3FCLs baseline architectures, while we trivially structured sub-SNNs from the baseline MLP by scaling its width by 1/10. We used direct encoding for all our experiments as it achieves higher accuracy [24]. For the KWS, we used a spectrogram of 90x40 as the input. In addition, for training, we used a maximum of 16 time steps to train all architectures. This number of steps is reduced for all architectures during inference. We trained all networks using BPTT with surrogate gradient functions, and used threshold-dependent batch normalization (tdBN) proposed in [37].

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

TABLE I
EVALUATED BASELINE SNN ARCHITECTURES

Operation	Latency (ns)	Energy (pJ)
Core operation* [34]	0.5	0.1
SRAM access** [35]	1.5	1.3
DRAM access [36]	60	30

* averaged for add, compare and branch instructions.

** averaged for different memory sizes.

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

Host computing systems: The energy and latency per instructions are modeled based on two simple hardware architectures, which allow us to evaluate our approach in terms of the number of memory operations and the efficiency of each memory operation. The first architecture has a single core and a main memory (first architecture in Fig 5). This architecture has only one type of memory operation, which always accesses data from the main memory. The second architecture is a temporal architecture (second architecture in Fig. 5), which consists of a simple core, an intermediate memory and a main memory. This architecture is similar to the previous architecture except that it has two different types of memory operations: (1) Inefficient memory operation that accesses the data from the main memory, and (2) efficient memory operation that accesses data from the intermediate memory. The utilization of the intermediate memory, which is proportional to the efficient memory operations, is modeled in our evaluation as output layer size. For example, in Algorithm 1, re-accessing v_0^L at the next spiking input occurs after accessing v_j^L for all j . This applies also to convolutional layers, except that a part of the output neurons (in every feature map) is accessed when an input neuron spikes. Table I shows the energy and latency models used for evaluation: the main memory was modeled as DRAM, the intermediate memory was modeled as an SRAM, and the other computations were modeled as an average core instruction.

B. Results

Accuracy: Table II presents the accuracy results of our experiments. For MNIST-MLP and KWS-3FCLs, the accuracy after aggregating the results of all sub-SNN is not affected compared the baseline architecture. This shows that for simple applications, the accuracy after sub-structuring the baseline architecture is maintained. For CIFAR10-VGG9 and CIFAR10-RESNET, the accuracy is degraded compared to the baseline architectures. The dataset used in these experiments is relatively more complex than the other datasets. Therefore, our results show that the accuracy of our approach degraded with increasing the complexity of the dataset. On the other hand, this degradation in

TABLE II
ACCURACY (%) ON DIFFERENT NETWORKS AND DATASETS

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

accuracy decrease when using more complex architectures (e.g., our approach has an accuracy of 80% for CIFAR10-RESNET and 73% for CIFAR10-VGG9, while the accuracies of baseline architectures almost remain constant). Finally, it is worth mentioning that, as shown in Table II, the average accuracy of all sub-SNNs before the aggregation function is close or even higher than the baseline accuracy. This implies that the aggregation function contributes to the degradation of the accuracy in some of the experiments.

Number of spikes per step: Fig. 6 shows the number of spikes per step for every layer. In our proposed approach, we calculated the number of steps for two types of execution schemes: (1) when all sub-SNNs are executed in parallel, and (2) when the sub-SNNs are executed sequentially. For parallel execution of sub-SNNs, the number of spikes per step for a specific layer is added for all sub-SNNs, while for sequential execution, the average is taken for every layer. Fig. 6(a), (c), (e) and (g) show that the number of spikes per step for the parallel sub-SNNs is on average similar to the baseline SNN; in some layers the number of spikes is higher for parallel sub-SNNs and lower for other layers. This means that the specialized SNNs require computing a similar number of features compared to the baseline SNNs. The sequentially executed sub-SNNs produce much lower number of spikes per step compared to the baseline SNN. This means that there is much lower number of operations needed by step. Fig. 6(b), (d), (f) and (h) shows the percentage of the spiking neurons per step (the number of spikes per step normalized by the layer width) for both the parallelly and sequentially executed sub-SNNs. On average, the proposed approach increases the number of active neurons per layer; this can be seen in most layers in Fig. 6(b), (d), (f) and (h). It is worth mentioning here that the number of spikes or the active neurons per layer does not precisely reflect the energy and latency. The energy and latency also depend on the structure of the SNN as will be shown next.

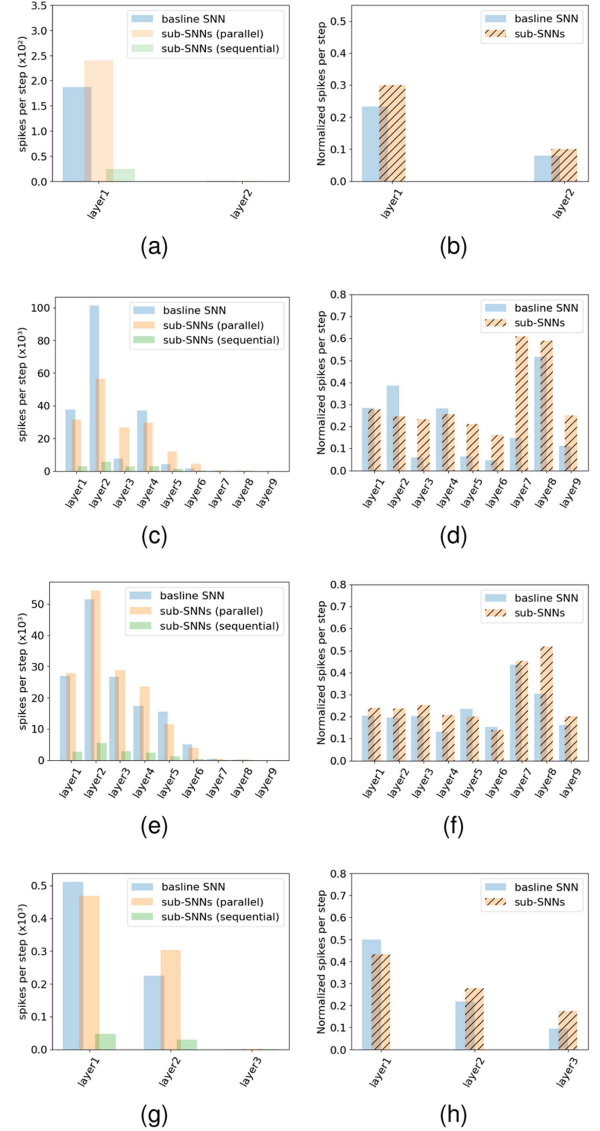


Fig. 6. Spikes per step: (a) MNIST-MLP, (c) CIFAR10-VGG9, (e) CIFAR10-RESNET, (g) KWS-3FCLs. Normalized spikes per step: (b) MNIST-MLP, (d) CIFAR10-VGG9, (f) CIFAR10-RESNET, (h) KWS-3FCLs.

Energy and latency: Figs. 7 and 8 show the energy and latency results when executing the baseline SNN and the sub-SNNs on a simple architecture without an intermediate memory and the same architecture with different sizes of intermediate memory. These results are shown in the logarithmic scale to be able to visualize the difference between the baseline SNNs and the sub-SNNs. The results show at least one order of magnitude gain in both energy and latency. The execution on an architecture with no intermediate memory shows the gain of our approach in terms of the number of memory operations. In addition, compared to the baseline SNN, the sub-SNNs make better utilization of an intermediate memory. This is because narrower layers have smaller reuse distances. E.g., a spike in an input layer requires updating all membrane potentials in the output layer. This means that a membrane potential at one layer is reused after updating the states of all membrane potentials in the same layer,

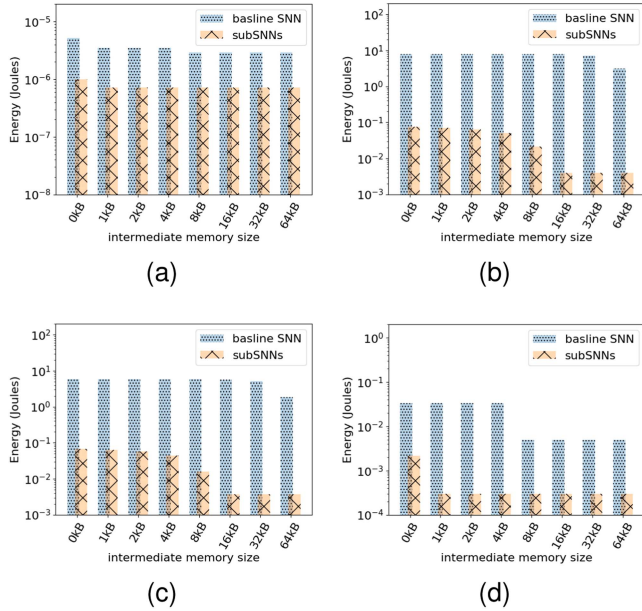


Fig. 7. Energy: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.

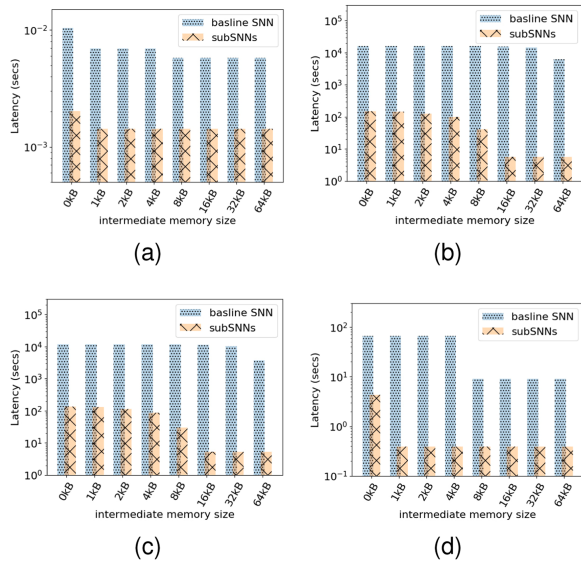


Fig. 8. Latency: (a) MNIST-MLP, (b) CIFAR10-VGG9, (c) CIFAR10-RESNET, (d) KWS-3FCLs.

making the reuse distance proportional to the layer’s width. Moreover, better utilization of the intermediate memory relaxes the requirements on the memory size; i.e., the runtime memory requirements).

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

intermediate memory of different sizes compared to the baseline SNN. This allows the proposed approach to relax the runtime memory requirements of the SNNs.

VI. DISCUSSION AND FUTURE WORK

Scaling application complexity: Our approach considerably reduces the memory overhead of SNNs; at least, for one order of magnitude for both energy and latency. On the other hand, our results show a limitation in approach when scaling the complexity of the application. This urge a need in future work to investigate the applications that can benefit from the improvements in energy and latency of our approach without sacrificing the accuracy. Another important aspect to investigate is the design of a more complex aggregation function that increases the accuracy without introducing a large computational overhead.

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

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

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

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

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

VII. CONCLUSION

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

REFERENCES

- [1] K. Roy, A. Jaiswal, and P. Panda, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, no. 7784, pp. 607–617, 2019.
- [2] X. She, S. Dash, and S. Mukhopadhyay, "Sequence approximation using feedforward spiking neural network for spatiotemporal learning: Theory and optimization methods," in *Proc. Int. Conf. Learn. Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=bp-LJ4y_XC
- [3] J. Kaiser, H. Mostafa, and E. Neftci, "Synaptic plasticity dynamics for deep continuous local learning (DECOLLE)," *Front. Neurosci.*, vol. 14, 2020, Art. no. 424.
- [4] N. Rathi et al., "Exploring neuromorphic computing based on spiking neural networks: Algorithms to hardware," *ACM Comput. Surv.*, vol. 55, no. 12, pp. 1–49, 2023.
- [5] B. Rueckauer, C. Bybee, R. Goetsche, Y. Singh, J. Mishra, and A. Wild, "NxTF: An API and compiler for deep spiking neural networks on Intel Loihi," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 3, pp. 1–22, 2022.
- [6] Y. Kim, Y. Li, A. Moitra, R. Yin, and P. Panda, "Sharing leaky-integrate-and-fire neurons for memory-efficient spiking neural networks," *Front. Neurosci.*, vol. 17, 2023, doi: [10.3389/fnins.2023.1230002](https://doi.org/10.3389/fnins.2023.1230002).
- [7] G. Neculae, O. Rhodes, and G. Brown, "Ensembles of spiking neural networks," 2020, *arXiv:2010.14619*.
- [8] N. Rathi and K. Roy, "STDP based unsupervised multimodal learning with cross-modal processing in spiking neural networks," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 5, no. 1, pp. 143–153, Feb. 2021.
- [9] D. Elbrecht, S. R. Kulkarni, M. Parsa, J. P. Mitchell, and C. D. Schuman, "Evolving ensembles of spiking neural networks for neuromorphic systems," in *Proc. IEEE Symp. Ser. Comput. Intell.*, 2020, pp. 1989–1994.
- [10] J. Shen, Y. Zhao, J. K. Liu, and Y. Wang, "HybridSNN: Combining bio-machine strengths by boosting adaptive spiking neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 9, pp. 5841–5855, Sep. 2023.
- [11] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv:1701.06538*.
- [12] C. Riquelme et al., "Scaling vision with sparse mixture of experts," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, vol. 34, pp. 8583–8595.
- [13] Z. Chen et al., "Mod-squad: Designing mixtures of experts as modular multi-task learners," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2023, pp. 11828–11837.
- [14] R. Vidya Wicaksana Putra and M. Shafique, "tinySNN: Towards memory- and energy-efficient spiking neural networks," 2022, *arXiv:2206.08656*.
- [15] Y. Kim, Y. Li, H. Park, Y. Venkatesha, R. Yin, and P. Panda, "Exploring lottery ticket hypothesis in spiking neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2022, pp. 102–120.
- [16] Y. Kim, Y. Li, H. Park, Y. Venkatesha, A. Hambitzer, and P. Panda, "Exploring temporal information dynamics in spiking neural networks," in *Proc. AAAI Conf. Artif. Intell.*, 2023, pp. 8308–8316.
- [17] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, 2017.
- [18] R. Yin, A. Moitra, A. Bhattacharjee, Y. Kim, and P. Panda, "SATA: Sparsity-aware training accelerator for spiking neural networks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 42, no. 6, pp. 1926–1938, Jun. 2023.
- [19] B. Na, J. Mok, S. Park, D. Lee, H. Choe, and S. Yoon, "AutoSNN: Towards energy-efficient spiking neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 16253–16269.
- [20] G. Datta, H. Deng, R. Aviles, and P. A. Beerel, "Towards energy-efficient, low-latency and accurate spiking LSTMs," 2022, *arXiv:2210.12613*.
- [21] Y. Kim, Y. Li, H. Park, Y. Venkatesha, and P. Panda, "Neural architecture search for spiking neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2022, pp. 36–56.
- [22] Y. Li, A. Moitra, T. Geller, and P. Panda, "Input-aware dynamic timestep spiking neural networks for efficient in-memory computing," 2023, *arXiv:2305.17346*.
- [23] Y. Kim and P. Panda, "Optimizing deeper spiking neural networks for dynamic vision sensing," *Neural Netw.*, vol. 144, pp. 686–698, 2021.
- [24] Y. Kim, H. Park, A. Moitra, A. Bhattacharjee, Y. Venkatesha, and P. Panda, "Rate coding or direct coding: Which one is better for accurate, robust, and energy-efficient spiking neural networks?," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2022, pp. 71–75.
- [25] A. J. Leigh, M. Heidarpur, and M. Mirhassani, "Selective input sparsity in spiking neural networks for pattern classification," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2022, pp. 799–803.
- [26] Y. Li and Y. Zeng, "Efficient and accurate conversion of spiking neural network with burst spikes," 2022, *arXiv:2204.13271*.
- [27] D. Auge, J. Hille, E. Mueller, and A. Knoll, "A survey of encoding techniques for signal processing in spiking neural networks," *Neural Process. Lett.*, vol. 53, no. 6, pp. 4693–4710, 2021.
- [28] J. K. Eshraghian et al., "Training spiking neural networks using lessons from deep learning," 2021, *arXiv:2109.12894*.
- [29] M. Zhang et al., "Rectified linear postsynaptic potential function for backpropagation in deep spiking neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 5, pp. 1947–1958, May 2022.
- [30] W. Gerstner and W. M. Kistler, "Mathematical formulations of Hebbian learning," *Biol. Cybern.*, vol. 87, no. 5, pp. 404–415, 2002.
- [31] E. Hunsberger and C. Eliasmith, "Spiking deep networks with LIF neurons," 2015, *arXiv:1510.08829*.

- [32] Q. Xu, Y. Li, J. Shen, J. K. Liu, H. Tang, and G. Pan, "Constructing deep spiking neural networks from artificial neural networks with knowledge distillation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2023, pp. 7886–7895.
- [33] R. Shwartz-Ziv and N. Tishby, "Opening the black box of deep neural networks via information," 2017, *arXiv:1703.00810*.
- [34] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
- [35] A. Pal, Y. Zhang, and D. D. Yau, "Monolithic and single-functional-unit level integration of electronic and photonic elements: FET-LET hybrid 6T SRAM," *Photon. Res.*, vol. 9, no. 7, pp. 1369–1378, 2021.
- [36] S. Kargar and F. Nawab, "Challenges and future directions for energy, latency, and lifetime improvements in NVMs," *Distrib. Parallel Databases*, vol. 41, pp. 163–189, 2022.
- [37] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, "Going deeper with directly-trained larger spiking neural networks," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 11062–11070.



Muath Abu Lebdeh received the B.Sc. degree in electronic engineering and the M.Sc. degree in computer engineering from Khalifa University, Abu Dhabi, UAE. He is currently working toward the Ph.D. degree with the Department of Information Engineering and Computer Science, University of Trento, Trento, Italy. His research interests include neuromorphic computing, energy-efficient systems, and emerging computer architectures.



Kasim Sinan Yildirim (Member, IEEE) is currently an Associate Professor with the Department of Information Engineering and Computer Science, University of Trento, Trento, Italy. His research interests include low-power and networked embedded sensing systems. He is interested in various aspects of such systems, including wireless sensing, embedded operating systems and runtimes, architectural support, and intermittent computing, and tiny machine learning.



Davide Brunelli (Senior Member, IEEE) received the M.S. (*cum laude*) and Ph.D. degrees in electrical engineering from the University of Bologna, Bologna, Italy, in 2002 and 2007, respectively. He is currently an Associate Professor of electronics with the Department of Industrial Engineering, University of Trento, Trento, Italy. He has authored or coauthored more than 280 research papers in international conferences and journals on ultra-low-power embedded systems, energy harvesting, and power management of VLSI circuits. He holds several patents and is annually

ranked among the top 2% of scientists according to the "Stanford World Ranking of Scientists" from 2020. His research interests include new techniques of energy scavenging for IoT and embedded systems, the optimization of low-power and low-cost consumer electronics, and the interaction and design issues in embedded personal and wearable devices. He is a member of several TPC conferences in the Internet of Things (IoT) and power management and is an Associate Editor for the IEEE.