2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00047

Skipper: Enabling efficient SNN training through activation-checkpointing and time-skipping

Sonali Singh
School of EECS
Pennsylvania State University
University Park, USA
sms821@psu.edu

Anup Sarma
School of EECS
Pennsylvania State University
University Park, USA
avs6194@psu.edu

Sen Lu
School of EECS
Pennsylvania State University
University Park, USA
szl5689@psu.edu

Abhronil Sengupta
School of EECS
Pennsylvania State University
University Park, USA
sengupta@psu.edu

Mahmut T. Kandemir
School of EECS
Pennsylvania State University
University Park, USA
mtk2@psu.edu

Emre Neftci
RWTH Aachen
Aachen, Germany
eneftci@uci.edu

Vijaykrishnan Narayanan
School of EECS
Pennsylvania State University
University Park, USA
vijaykrishnan.narayanan@psu.edu

Chita R. Das
School of EECS
Pennsylvania State University
University Park, USA
cxd12@psu.edu

***Abstract*—Spiking neural networks (SNNs) are a highly efficient signal processing mechanism in biological systems that have inspired a plethora of research efforts aimed at translating their energy efficiency to computational platforms. Efficient training approaches are critical for the successful deployment of SNNs. Compared to mainstream deep neural networks (ANNs), training SNNs is far more challenging due to complex neural dynamics that evolve with time and their discrete, binary computing paradigm. Back-propagation-through-time (BPTT) with surrogate gradients has recently emerged as an effective technique to train deep SNNs directly. SNN-BPTT, however, has a major drawback in that it has a high memory requirement that increases with the number of timesteps. SNNs generally result from the discretization of Ordinary Differential Equations, due to which the sequence length must be typically longer than RNNs, compounding the time dependence problem. It, therefore, becomes hard to train deep SNNs on a single or multi-GPU setup with sufficiently large batch sizes or time-steps, and extended periods of training are required to achieve reasonable network performance.**

In this work, we reduce the memory requirements of BPTT in SNNs to enable the training of deeper SNNs with more timesteps (T). For this, we leverage the notion of activation re-computation in the context of SNN training that enables the GPU memory to scale sub-linearly with increasing time-steps. We observe that naively deploying the re-computation based approach leads to a considerable computational overhead. To solve this, we propose a time-skipped BPTT approximation technique, called *Skipper*, for SNNs, that not only alleviates this computation overhead, but also lowers memory consumption further with little to no loss of accuracy. We show the efficacy of our proposed technique by comparing it against a popular method for memory footprint reduction during training. Our evaluations on 5 state-of-the-art networks and 4 datasets show that for a constant batch size and time-steps, *skipper* reduces memory usage by 3.3× to 8.4× (6.7× on average) over baseline SNN-BPTT. It also achieves a speedup of 29% to 70% over the checkpointed approach and of 4% to 40% over the baseline approach. For a constant memory budget, *skipper* can scale to an order of magnitude higher timesteps compared to baseline SNN-BPTT.

***Keywords*-SNN; BPTT; compute and memory.**

I. Introduction

Neuromorphic hardware implementing SNNs has the potential for low-latency and energy-efficient signal processing due to their locally-dense and globally-sparse architecture [1], [2], [3]. Neuromorphic hardware is particularly suited for processing the output of neuromorphic sensors [4] that produce streams of events rather than frames. Prior research has shown that event-based vision cameras can be very useful in solving depth estimation and optical flow problems using deep neural networks [5]. Also, several AI applications, realized using SNNs and benchmarked on prototype neuromorphic hardware [2], have demonstrated considerable performance and energy benefits compared to mainstream platforms [6], establishing the effectiveness of such platforms. Although such neuromorphic hardware platforms are rapidly evolving and are highly efficient at processing spiking temporal signals, they are still limited to inferences [7] or constrained online learning [2] scenarios, and general-purpose systems such as GPUs and CPUs continue to be the predominant platforms for training of large-scale SNNs.

Most SNNs implement a variant of the integrate and fire (I&F) neuron [8], which captures the neuron's temporal dynamics and its all-or-none activation. The discretization of the I&F neuron differential equations results in a special case of recurrent neural network (RNN), with two key challenges: firstly, the activation function is a non-differentiable step function, and secondly, the typical sequence length is considerably longer than RNNs to account for the temporal dynamics. The first challenge can be addressed using the surrogate gradient method, which assumes a smooth surro-

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00047

gate network for the purposes of differentiation [9]. This enables the use of stochastic gradient descent to optimize task-relevant loss functions, and therefore the use of auto-differentiation tools in ML frameworks [10], [11].

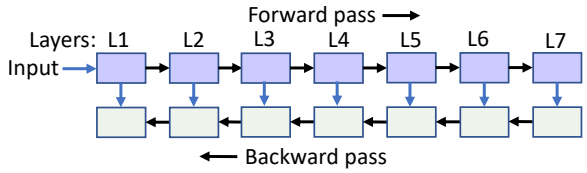
Gradient descent applied to SNNs is a variant of the BPTT algorithm [12], which scales as the number of neurons times the number of time steps (typically in the hundreds). The dependence on the latter makes SNN training particularly memory- as well as compute-intensive. This high complexity makes SNN training a major obstacle for real-world application scenarios [13]. To address this problem, several algorithmic techniques have been proposed, such as approximations of the real-time recurrent learning (RTRL) rule [14], [15], [16], which eliminates the time dependence of the network. Others have proposed local loss functions [14] and feedback alignment [17], which eliminates the need to back-propagate the loss across layers and are less compute- and memory-intensive than BPTT. Other techniques such as ANN-to-SNN conversion [18], [19], [20] and STDP-based unsupervised learning [21], [22] have also been proposed. However, approximations to RTRL and local learning result in worse accuracy compared to BPTT [23], [24] and so do conversion and STDP-based mechanisms. BPTT, thus, remains the gold standard for training SNNs, and improving its efficiency in terms of reduced memory footprint and execution time on current compute platforms is key to its adoption in large-scale, real-world applications.

Motivated by these observations, in this work, we present two complementary techniques to minimize the memory and computation overheads of BPTT-based SNN training on a GPU. In order to alleviate the high memory cost of SNN training that scales linearly with time steps, we first leverage a well-known technique, called *activation checkpointing*, where intermediate activations in a multi-layer network are not saved, to reduce the memory demand. This concept (also referred to as *gradient checkpointing* in deep learning literature), has been applied for training deep neural networks (DNNs) on GPUs, where memory is often a limiting factor, by saving a subset of the layer activations instead of the activations of all the layers (as is usually the case), and then re-computing them when required [25], [26]. The recomputation costs another 30%-35% increase in training time [25], which we mitigate by proposing a computation skipping mechanism.

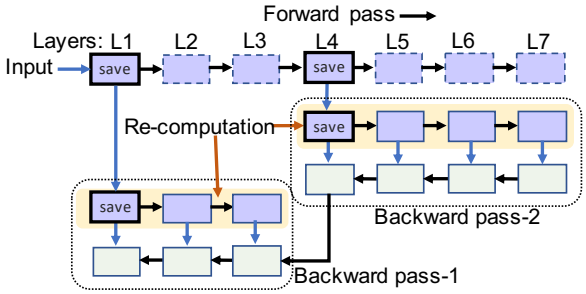
While gradient checkpointing has been applied in the context of CNNs and RNNs, we establish activation checkpointing in SNNs which serves as a foundation for our subsequent innovations. Thus, we first apply checkpointing to SNNs in the *temporal* dimension i.e. instead of saving all the intermediate neural states, we save only a subset of these (activations) and then recompute them at the time of backpropagation. Second, we propose a novel technique viz. **activation checkpointing with time-skipping** or ***skipper*** to *alleviate the computational overhead* of checkpointing, by using a Spike Activity Monitoring (SAM) approach that reduces the training time compared to baseline SNN-BPTT and yields additional memory savings compared to only checkpointing. In summary, following are our *contributions*:

- We apply checkpointing to SNN in the form of activation checkpointing to reduce its memory consumption during training by trading off additional computations for memory savings. We build a framework on top of a popular deep learning library to readily train SNNs with BPTT at a reduced memory cost. This can be used by researchers for exploring deeper SNNs to solve complex tasks.
- We propose a novel technique called *Skipper*, which approximates BPTT (considered as our baseline) using a Spike Activity Monitoring (SAM) approach and augments activation checkpointing to not only alleviate its computational overhead, but also to reduce its training memory consumption further, with little to no loss of accuracy. The computation skipping in *skipper* is achieved by intelligently utilizing the neuron computation dynamics in the form of spike activity during the forward pass.
- We use our checkpointing framework to evaluate the efficacy of both activation checkpointing and *skipper* on 5 SNN workloads with 4 different vision datasets. Experimental results show that for constant timesteps and batch size, activation checkpointing can save up to 4.3× (4.2× on average over evaluated SNNs) the amount of memory compared to the baseline, while incurring ≈35% computational overhead on average. Our proposed activation checkpointing with time-skipping technique (*skipper*) further improves memory savings by up to ∼2× compared to plain checkpointing, while also saving compute time by 29% to 70% across the 5 workloads. This translates to a memory saving of up to 8.4× (6.7× on average) and 4% to 40% run-time speedup compared to baseline BPTT.
- We compare our techniques with truncated back-propagation-through-time (TBPTT [27]), a widely used technique to reduce memory consumption while training RNNs and demonstrate the advantages of our approach.
- We also compare our techniques with a prior work [28] and demonstrate the scalability of our approach.
- Finally, we show the efficacy of our approach on the neuromorphic vision datasets DVS-gesture and N-MNIST, highlighting its applicability to both frame-based and event-based data.

We perform all our experiments on GPUs for ease of programmability and reproducibility, but the techniques proposed in this work are agnostic to the underlying platform and therefore equally applicable to future/custom neuromorphic hardware platforms as well.



(a) All activations stored (baseline approach)



(b) Selected activations stored (checkpointing approach)

Figure 1: Typical training process of a DNN.

II. Related Work

A number of recent research proposals have designed neuromorphic hardware for efficient SNN inference and on-chip learning through the use of innovative devices and circuits [29], micro-architecture [30], [31], dataflow [32], and interconnect [2], [7]. However, there is a relative lack of prior works aimed at making the existing training techniques more computationally efficient. To fill this gap, we optimize SNN training by investigating the "compute-memory tradeoff" hitherto unexplored in the SNN context. Although this tradeoff in the form of activation (gradient) checkpointing has been studied in the context of general computation graphs [33], they have only recently been applied to backpropagation in DNNs. One of the earliest works to explore gradient checkpointing in DNNs, [25] proposed algorithms to reduce the memory consumption of n-layered DNN by $\sqrt{n}$ and in the extreme case by $O(nlogn)$. Ref. [26] proposed a dynamic programming-based approach to reduce the memory consumption of BPTT when training RNNs given a fixed memory budget. Authors in [34] present a recomputation-based method that can be applied to a wider range of neural network topologies (such as those with long skip connections). However, all of these methods incur a computational overhead of at least one additional forward pass. Additionally, our proposed techniques are orthogonal to [34], as we apply checkpointing to the temporal dimension, which makes it agnostic to the network topology.

To the best of our knowledge, *no other works have attempted to reduce the computational overhead of the extra forward pass.* Probably, the most closely related work to ours is [28], which reduces the memory footprint of BPTT in SNN through a combination of temporal truncation (TBPTT) and locally-supervised layers. However, they simplify the complexity of the training algorithm in space, whereas *skipper* does so in time. Besides, the scalability of their approach to deeper networks (such as ResNets) is not clear.

We now briefly discuss activation checkpointing in DNNs, in which the idea is to only selectively store the activations from the forward pass, instead of storing the activations of all the layers. The baseline technique is shown in Figure 1(a) – where all the activations from the forward pass are stored in memory. During the backward pass, the stored activations are recalled for gradient computation. However, in the case of Gradient Checkpointing, only a few layer activations are stored (marked as 'save' in Figure 1(b)). Thus, the backward pass now runs in multiple passes to complete gradient computation of the entire network. As shown in the figure, in Backward pass-2, activations are restored by re-computing the discarded states from L5 onwards, which are then consumed in the gradient computation process. In the subsequent pass, the remaining activations are restored (L2 to L3) and the gradient computation process is completed for the corresponding network segment. The number of passes required to complete the BP thus equals the number of checkpointed states.

III. Background

In this section, we provide a brief overview of how SNN training is performed using gradient descent-based optimization techniques and the issues involved therein, which serve as the main motivation for our current work.

A. The Neuron Model

The neuron model describes the internal state update dynamics as well as the firing behavior of a spiking neuron. Although several neuron models exist in the literature that replicate biological neurons at varying levels of bio-plausibility [35], variations of the linear leaky-integrate-and-fire (LIF) neuron are widely used in ML-based optimizations due to their simplicity and scalability. A discrete-time formulation of the LIF neuron suitable for digital simulations is characterized by the following equation:

$$U_t^l = \lambda U_{t-1}^l + W^l o_t^{l-1} - \theta o_{t-1}^l, o_{t-1}^l = \begin{cases} 1, & \text{if } U_{t-1}^l > \theta \\ 0, & \text{otherwise} \end{cases}, \quad (1)$$

where U_t^l is the neuron's membrane potential (internal neuron state) at time t, l is the layer index, λ (≤ 1) is the membrane potential leak, W_l is the weight matrix connecting layers $l-1$ and l, o_t is the spike vector at time t, and θ is the firing threshold potential. The first term on the right hand side of equation 1 carries forward the neuron state from previous to the current time-step (modulated by leak λ); the second term is a weighted sum of spikes coming from the previous layer (a feed-forward connection); and the third

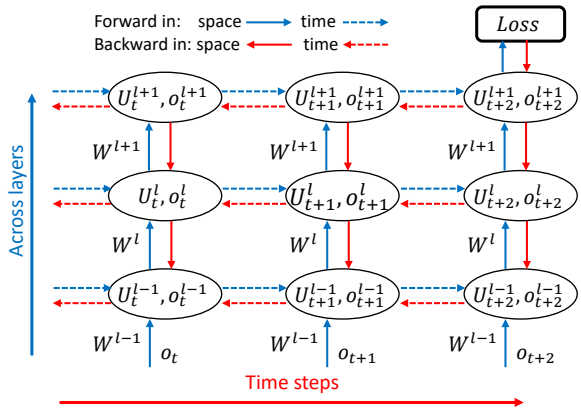


Figure 2: Back-propagation through time in SNNs. The vertical dimension shows different layers of an SNN, while horizontal dimension shows unrolling in temporal dimension.

term arises from the thresholding non-linearity that decreases the membrane potential by θ if an output spike o^l_{t-1} is generated by a neuron (a recurrent connection). The SNN can, thus, be considered a specialized form of a Recurrent Neural Network (RNN) [12], [9].

B. SNN training via backpropagation through time

BPTT consists of minimizing a global loss function L by applying gradient backpropagation to the network unrolled in space and time as depicted in Figure 2. For a loss L computed at time t, the gradient descent weight update ΔW for layer l is governed by:

$$\Delta W^l = -\eta \frac{\partial L_t}{\partial W^l} = -\eta \sum_{s=0}^{t} \delta^l_s {o^{l-1}_s}^\top,$$
$$\text{where } \frac{\partial L_t}{\partial U^l_s} = \delta^l_s = \text{diag}(\sigma'(U^l_s)) {W^{\top,l+1}} \delta^{l+1}_{s+1} + \lambda \delta^l_{s+1}. \tag{2}$$

σ' is the smooth surrogate function for computing the gradient of the activation function [9]. Note that the reset term is not taken into account for the gradient computation.

C. Truncated backpropagation through time

Truncated backpropagation through time (TBPTT) [27] is a popular technique in deep learning literature that is often used to reduce the memory consumption of RNN and its variants during training. In its simplest form, the network is unrolled for a shorter computation window also known as the truncation window $t' < T$. A loss value is calculated at t', its derivatives are backpropagated through the network and the weight gradients are computed and stored. At this point, the computation graph of the network is discarded and the corresponding memory is released. This process of loss calculation and backpropagation is repeated for multiple truncation windows, and the weight gradients calculated at time $(t', 2t', ..., T)$, are summed to get the final gradient, which is then used by the optimizer to update the network parameters.

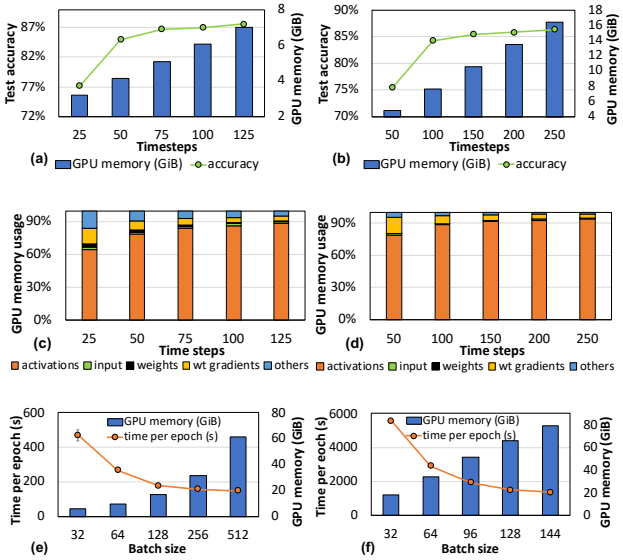


Figure 3: SNN accuracy and training memory consumption vs timesteps for (a) VGG5, CIFAR10, (b) ResNet20, CIFAR10. Breakdown of GPU tensor memory occupancy vs timesteps at B=32 for (c) VGG5, (d) ResNet20. Training time per epoch and GPU memory consumption vs batch size for (e) VGG5 and (f) ResNet20.

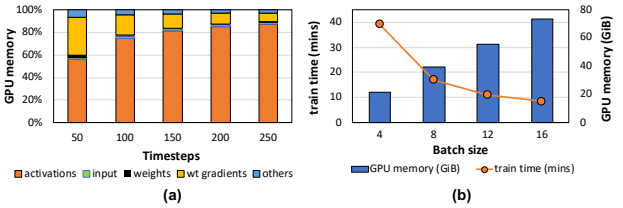


Figure 4: Training ResNet34 SNN on ImageNet via BPTT (a) GPU memory breakdown vs timesteps at B=1, and (b) Time to train 800 samples in a data-parallel regime on 4 A100 GPUs and corresponding memory (per GPU) vs batch size.

IV. Motivation

In this section, we experimentally demonstrate the reasons for the highly memory-intensive nature of SNN training. Figures 3(a) and (b) show the SNN accuracy on the testing set when trained with an increasingly large time-window for a small (VGG5 [36], [37]) and a large network (ResNet-20 [38], [19]), respectively, on the CIFAR-10 dataset [39]. The network test accuracy improves with more timesteps

(left axis, Figure 3(a) and (b)). This, however, comes at an increased memory cost that scales linearly with the number of timesteps, as plotted on the right axis of Figure 3(a), (b).

To investigate further, Figures 3(c) and (d) plot the relative memory consumption of different types of tensors present on the GPU during training of VGG5 and ResNet20 SNNs as a function of timesteps for batch size 32. These measurements were taken after the GPU reaches its maximum memory occupancy and becomes steady (i.e., from the second iteration onwards), at which point there is dedicated memory allocated to all types of tensors required for training. Figures 3(c) and (d) categorize these tensors as input, model, activations, optimizer, and others. The input tensor consists of spiking data to be supplied as input to the SNN and the corresponding labels. The model tensor represents the trainable network parameters. The optimizer portion jointly consists of the weight gradients and gradient moments that depend on the type of optimizer used, as well as some non-trainable parameters (e.g. leak, threshold etc). Since we have used the Adam optimizer [40], the gradient moments are $2\times$ the size of the weights, whereas the weight gradients are of the same size as the weights. Nevertheless, we observe that a major portion of the tensor memory (60% - 95%) is occupied by activations, which consist of time-dependent spikes and neuronal states. This is partly due to the fact that large batch sizes are used in training to obtain higher GPU throughput[1]. However, the relative proportion of activations also increases with the number of timesteps for a constant batch size. This is because, the time-wise activation tensors are saved on the GPU memory during the forward pass computations since they are required during the backward pass computations as per equation 2. To further emphasize this point, we report the GPU tensor memory breakdown while training ResNet34 on ImageNet against timesteps for a batch size of 1 in Figure 4(a). The time-wise activations account for 56% to 90% of the total memory consumption, thus establishing SNN activations as a significant memory bottleneck.

We also report the training latency and memory consumption required to train this SNN in a data-parallel regime on 4 A100 GPUs as a function of batch size in Figure 4(b), with B=16 being the largest batch size that can fit on the GPUs at T=200. At B=16, the time to train on 1M ImageNet training samples would be $\sim$3.5 days (extrapolated from 10 minutes for 800 samples) for a single epoch. Even if we perform transfer learning and train the SNN only for 20 epochs, this would translate to roughly 70 days of training, assuming we have all the hyper-parameters properly tuned. The huge computational complexity of training not only arises from the temporal nature of SNNs but also from the discretization function, which requires them to be simulated for 100s of timesteps, as noted earlier. This observation presents us with a challenge as well as an opportunity to reduce the training memory consumption of SNNs by directly lowering its activation memory footprint.

The potential memory savings could be utilized in the following ways: **(i)** for explorations of deeper networks with longer temporal horizons (demonstrated in Figure 14 in results section), **(ii)** to train SNNs on larger mini-batches, thus improving GPU throughput and speeding up training (shown in Figure 11 in results section). Figures 3(e) and (f) show the time required for a single training epoch as a function of the batch size. The training time decreases by $\sim 5\times$ as we increase the batch size from 32 to 512 for VGG5 and from 32 to 144 for ResNet20. This comes at the cost of increased memory requirement that also scales linearly with batch size. **(iii)** to enable multiple simultaneous trainings on the GPU, often useful in hyper-parameter search/tuning.

V. Activation checkpointing in SNN

As explained earlier in Section III-B, the SNN forward pass computations are completely unrolled in time and the resulting intermediate states (U_t^l, o_t^l) are saved in memory for the backward propagation of error gradients to take place. This model serves as our baseline. Figure 5 shows the computational graph of an SNN being trained for $T = 20$ timesteps. Figure 5(a) shows the layer-to-layer interactions for a single timestep and 5(b) shows the unrolled timesteps for a single layer l. In order to reduce the memory, we propose to apply activation (gradient) checkpointing to this computational graph, i.e., *we drop some of the intermediate neural states and recompute them later during the backward pass, at the cost of an extra forward pass computation*. For a given T, the amount of memory saved will depend on the number of times the network is checkpointed, denoted as C in the rest of this paper. Figure 5(b) depicts this idea more clearly through the following example. In this, the SNN from Figure 5(a) is checkpointed twice ($C = 2$).

Step1: The forward pass computation takes place and after $t = 19$, the loss function L and its derivative at $t = 19$ are computed. The time-wise activations are only saved twice at $t = 0$ and $t = 10$, and therefore, the corresponding memory occupancy is only twice the activation size per timestep, at this stage.

Step2: The network is unrolled from the most recent activation checkpoint (at $t = 10$) and the intermediate states are *recomputed*. To be precise, the forward pass computations are performed a second time from $t = 11$ to $t = 19$ and the corresponding states are saved in memory. The activation memory at this point comprises of the activations from Step1 along with this unrolled time-segment of intermediate states.

Step3: Back-propagation takes place from $t = 19$ to $t = 10$ and the corresponding error gradients up until $t = 10$ $(\frac{\partial L_{10}}{\partial W^l})$ are computed as per the chain rule. The

[1]Another reason for this is that CNNs have fewer trainable parameters per activation than fully connected networks.

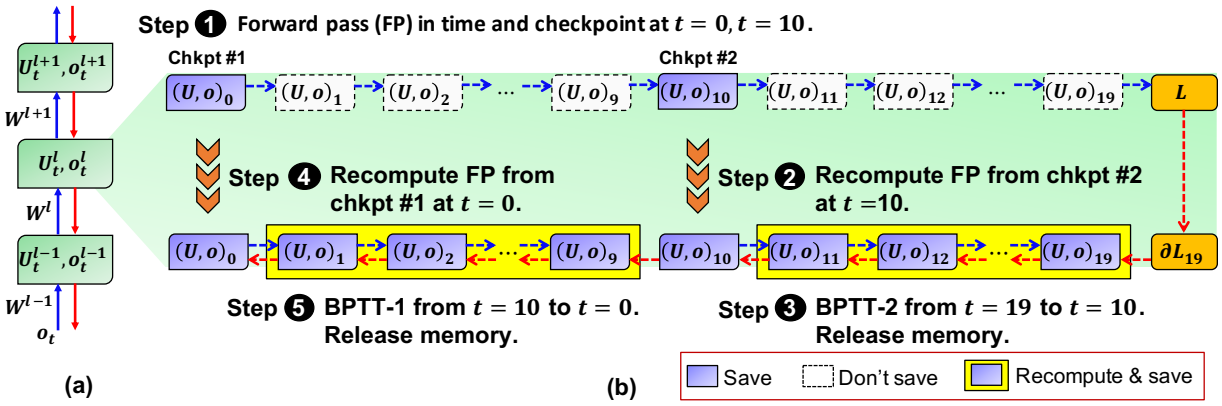


Figure 5: (a) SNN inter-layer spike interactions per time step. (b) Activation checkpointing shown in layer l applied along temporal dimension. Evolution of states (U_t^l, o_t^l) follows equation 1 and backward pass follows equation 2.

device activation memory corresponding to these time-steps is released.

Step4: Step2 is repeated for the next most recent checkpoint (at $t = 0$) and the intermediate states from $t = 1$ to $t = 9$ are re-computed in the forward pass.

Step5: The error gradients calculated in Step3 ($\frac{\partial L_{10}}{\partial W^l}$) are used to back-propagate the errors from $t = 10$ to $t = 0$. The network weight updates (ΔW) are then calculated as a sum of these error gradients for all time-steps, as per equation 2 and the corresponding activation memory is freed. Note that for each training iteration, the recomputation for each time step happens only once.

A. Tradeoff between the number of time steps, checkpoints and layers

For a DNN with n layers, [25] has shown that the memory cost can grow as $O(\sqrt{n})$, instead of $O(n)$, at the cost of an extra forward pass. When applied to SNNs along the temporal dimension, the activation memory consumption will be a function of T and C. Specifically, for an SNN checkpointed C times and simulated for T timesteps, $(C \leq T)$, the size of each time-segment will be T/C. As a result, the total activation memory requirement for training can be expressed as follows:

$$totalActCost = O\left(\frac{T}{C}\right) + O(C). \tag{3}$$

Thus, the memory cost of running SNN backpropagation on each time-segment is $O(T/C)$. Note that the second part of the above equation is the memory required to save the intermediate states between time-segments. As per this algorithm, the activation memory cost is minimized at $C = \sqrt{T}$ [25].

Further, in the case of SNNs, we cannot have an arbitrary number of checkpoints C. This is because, for a baseline SNN with L_n layers, we must have $T > L_n$ for spikes to propagate to all the layers after the thresholding function has been applied. Correspondingly, for checkpointed SNN, the length of *each* time segment (T/C) must be greater than L_n, for the propagation of information to all the layers in that time segment (i.e. $T/C > L_n$). As a result, C is upper bounded by the ratio T/L_n of an SNN.

B. Computational overhead of checkpointing

In a typical single GPU neural network training, the forward and backward passes constitute roughly $1/3^{rd}$ and $2/3^{rd}$ of the total number of computations per iteration. This is because, for each layer, the backward pass performs two sets of computations – one for computing the error gradients with respect to the activations and another with respect to the weights. As a result, the activation checkpointing is expected to incur $\sim$33% increase in training time.

Next, we discuss the opportunity for reducing the computational overhead of checkpointed SNN to leverage benefits in terms of overall computation time, in addition to memory footprint reduction.

VI. Overcoming the Computation Overhead of Activation Checkpointing through Skipper

Although activation checkpointing in SNNs is a promising technique to reduce its memory cost during training, it incurs an increased computational overhead of $\sim$33%, which can be considerable for long-running and large-scale SNN training. In this section, we explore techniques to alleviate this extra computational cost without compromising on the memory benefits. Towards this end, we re-examine the computation graph created during training of a checkpointed SNN and notice that the same forward pass computation is performed twice (Steps 1, 2 and 4 in Figure 5(b)). Our experience working with spiking data for vision-based applications has shown that some of the spike patterns can be redundant across time steps and it could be worthwhile to simply skip these computations to reduce training time. As a result, the

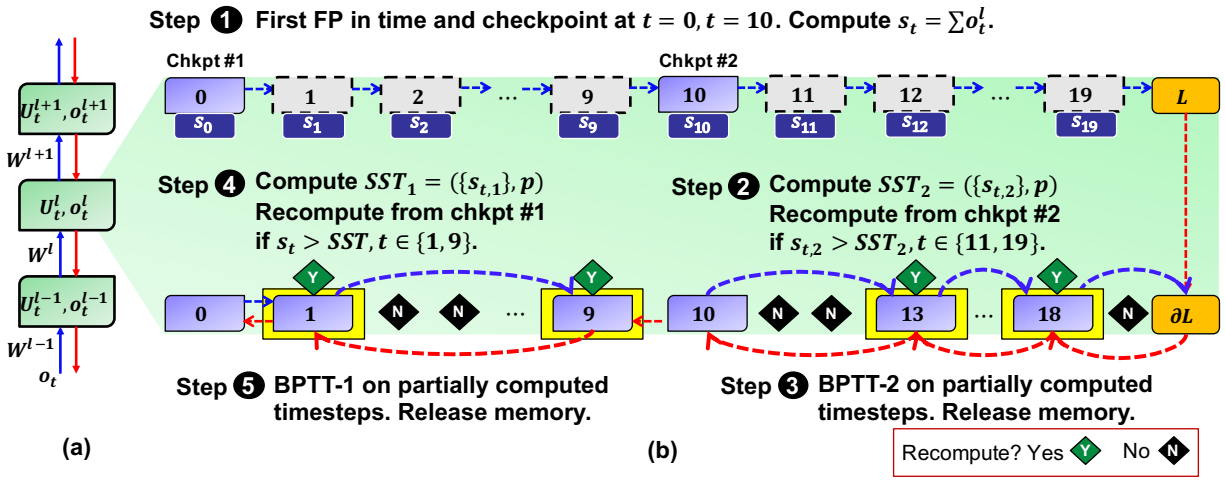


Figure 6: (a) SNN inter-layer spike interactions per time step. (b) An example schematic of *skipper*. Neuronal states (U_t^l, o_t^l) are represented by t for readability. Evolution of states follows equation 1 and s_t are simultaneously saved. Recomputation at time t and the corresponding backward pass are performed if $s_t > SST$.

corresponding backward pass computations would also not take place. This is because, in the case of training with auto-differentiation tools such as Pytorch [10] or Tensorflow [11], the tool builds a computation graph by reading the forward pass description of a network and imperatively traverses the graph symmetrically in the reverse order during the backward computation, thus implicitly realizing the chain-rule of differentiation. However, if we blindly avoid calculating some of the intermediate states in the forward pass, the corresponding gradients for those time-steps will not be computed in the backward pass, and this could affect network convergence during training.

In the case of activation checkpointing, however, the fact that the forward pass graph is traversed twice is an opportunity that can be leveraged to drop some computations without hurting network performance. Our key insight, therefore, is to collect some information about the neural dynamics during the first forward pass and use it to *intelligently skip* or drop some computations during the second forward pass, and thus recover some of the computational overhead of activation checkpointing. The challenge is to identify such a *metric* that is low overhead in terms of memory and computation time and yet powerful enough to convey critical information about whether or not to skip computations during the second forward pass, succinctly.

A. Skipping computation based on temporal spike activity

To avoid network convergence issues, computations cannot be skipped arbitrarily. Further, since we want to skip computations at the time granularity, which essentially means avoiding forward pass computations for all layers at that time-step, our decision-making metric needs to be a time-dependent variable. The neuron membrane potential and output spikes (U_t^l, o_t^l) are possible candidates for such a decision-making process as they indicate the current level of activity in the network. Of these, we choose the output spikes o_t^l as our indicator since the all-or-none behavior of spikes is an intrinsic yardstick of the current activity level in the network. We take the sum of the spikes as a heuristic to gauge the current activity level at every timestep. Further, all layers in the network simultaneously emit spikes at every timestep. We, therefore, compute and save the sum of spikes from each of the layers at a timestep in the forward pass (denoted as s_t in Figure 6(b)) and expressed as follows:

$$s_t = \sum_{l=0}^{l=L_n} sum(o_t^l) \tag{4}$$

Thus, we propose a Spike Activity Monitoring (SAM) mechanism which computes and saves s_t for every timestep during the first forward pass of the network. Next, we propose a metric, called *Spike-Sum-Threshold*, which is calculated for each of the checkpoints just before the second forward pass (denoted as SST_c, where c is the checkpointed segment number). In *skipper*, we measure this value by taking the p^{th} percentile of the spike sums corresponding to each checkpoint.

$$SST_c = percentile(\{s_{0(c)}, s_{1(c)}, ..., s_{T/C(c)}\}, p) \tag{5}$$

where p can be treated as an additional hyper-parameter in the training process.

During the backward pass of training, which now involves re-computation of 'non-checkpointed' activations, SAM is used to compare the the spiking activity of the current timestep (s_t) against the corresponding SST value. If

the spiking activity is found higher than this *Spike-Sum-Threshold* ($s_t > SST_c$), then re-computation takes place as usual for the particular time-step, otherwise, it is skipped.

Figure 6(b) depicts this idea pictorially in 5 steps, and we continue using the same example configuration as the previous section.

Step1: The forward pass computations are performed in a similar fashion to Step1 of activation checkpointing technique, i.e., only the intermediate states at $t = 0$ and $t = 10$ are saved. In addition to the computation of intermediate states, the output spikes generated from all the layers at that time-step are summed up and stored. Thus, a spike-count is calculated per time-step (shown as $s_0, ..., s_{19}$ in the figure), which serves as a metric for indicating the spike activity at that time-step.

Step2: Prior to recomputing the intermediate states for the backward pass, we perform a look-up on the Spike Activity Monitor and for the spike-counts corresponding to the latest checkpoint's time-segment, we calculate its p^{th} percentile as per equation 5, which serves as our *Spike-Sum-Threshold* (SST_c). We then compare each $s_{t(c)}$ with this SST_c and skip the computation of those timesteps whose s_t is lower than this threshold value. The percentile p is critical to our technique as it decides the number of time-steps that can be skipped. Higher p values will lead to higher *Spike-Sum-Threshold*s (equation 5). As a result, there will be more timesteps at which spike-counts s_t will fail to cross this threshold and thus more intermediate steps can be skipped.

Step3: Since we drop some of the compute-iterations during the forward pass in Step2, a shallower computation graph is created in memory, as shown in the Figure 6(b) (steps 2 and 3). The backward computation will now traverse through this new and shorter graph of checkpointed segment #2 at the end of which, the corresponding memory will be released.

Step4: We repeat Step3 for the next most recent time-segment, starting from $t = 0$ and selectively re-compute the intermediate states depending on the SST_c value.

Step5: Similar to Step4, we perform a backward pass on the time-skipped checkpointed segment #1.

To summarize, memory gets allocated every time a forward pass takes place and gets deallocated during the backward pass. In the case of SNN, the forward pass activation memory also consists of time-unrolled spikes and neural states. As a result, the allocated memory depends on the level of temporal unrolling. It is to be noted that *skipper* saves memory by unrolling fewer *non-sequential* timesteps during the second forward pass. The intermediate timesteps that are not a part of this new unrolled graph are essentially 'skipped', thus also saving compute time.

Choice of Spike Activity Monitor: Although we use the sum of spikes across all layers per timestep as a low overhead Spike Activity Monitor due to its simplicity, it is possible to use other metrics that can monitor network activity at a finer granularity, e.g., the sum of spike counts weighted by the neuron count in each layer, the $l2$-norm of neuron trace per timestep, or a combination of both. The impact of more sophisticated activity monitoring mechanisms on $skipper$'s accuracy is an interesting future research direction.

B. Impact of skipping computations

Here we discuss the impact of computation skipping from a theoretical standpoint by analyzing the key parameters T (no.of timesteps), C (no. of checkpoints), L_n (no. of layers) and p (percentile) during training. Firstly, with *skipper*, there is a reduction in computation time which directly depends on the number of time-steps skipped. In addition, the memory cost of training also reduces in proportion to the skipped time-steps. The new activation memory requirement for training with *skipper* can be expressed as:

$$totalActCost_{skipper} = O\Big((1 - \frac{p}{100}) \times \frac{T}{C}\Big) + O(C) \quad (6)$$

Thus, we can skip a larger fraction of the timesteps if the number of checkpoints C is low or if T/C (length of a checkpointed time-segment) is high. Another constraint applicable specifically to SNNs is that the number of timesteps that are *not* skipped per checkpoint must now be greater than the number of layers, i.e., $(1 - \frac{p}{100})\frac{T}{C} > L_n$, which yields the following upper bound on the fraction of time-steps that can be skipped for a given T, C and L_n:

$$\frac{p}{100} < 1 - \Big(\frac{C}{T/L_n}\Big) \quad (7)$$

This, combined with the fact that $C < \frac{T}{L_n}$ (from Section V-A) together provide the boundary conditions for setting C and p for a constant T and L_n. Equation 7 thus provides a simple rule of thumb for determining the maximum %age of time-steps that can be skipped (p) for an SNN with a particular T, C and L_n. Thus, we can skip more time-steps for networks with a large $\frac{T}{L_n}$ ratio or smaller C, implying that for a constant $\frac{T}{L_n}$ ratio, if an SNN is checkpointed many times, the scope for skipping time-steps and thus reducing training time decreases.

Note that, due to skipped time-steps, the functional outcome of *skipper* will be different from the vanilla checkpointing and baseline BPTT algorithms, and therefore, involves a trade-off in terms of accuracy, in addition to the memory and computation metrics discussed above. As a result, we set the time-skipping percentage p at a value that minimally impacts network accuracy. For this, we start with the theoretically maximum p (Equation 7) and slowly decrease it, while measuring its impact on network accuracy for a few training iterations each time. We stop at the p value where the accuracy loss is minimal.

In our study, we extensively evaluate the trade-offs involving the accuracy, memory and latency metrics, the details of which are discussed in the subsequent sections.

VII. Evaluation and Results

We implemented all the presented ideas including baseline and truncated BPTT using the PyTorch framework [10] running on a GPU, in an end-to-end fashion. Thus, performance metrics (with respect to memory, compute and accuracy) are directly measurable at a system level. Specifically, we first implemented activation checkpointing for SNN as a generalized framework that can be used to checkpoint and reduce the memory consumption of any feed-forward SNN during training. We then implemented activation checkpointing with time-skipping ($skipper$) in which the network can dynamically skip some percentage of the time-steps (a user-defined parameter) based on its activity in the first forward pass. The impact of *skipper* is also directly observed during training in terms of lower run-time and memory consumption as reported in the next section.

We evaluate our techniques by training SNNs to solve the image classification and action recognition problems on 3 small (VGG5, LeNet, custom-Net) and two large network topologies (VGG11 and ResNet20) in conjunction with 4 different datasets viz. CIFAR10 & CIFAR100 [39], DVS-Gesture [41] and N-MNIST [42] – the latter two being event-based datasets, specifically tailored for neuromorphic processing. For the evaluation of network performance in terms of accuracy, we train all the networks end-to-end for 20 epochs as per the hybrid training method described in [37] using the ADAM optimizer. As per this technique, we pre-initialize the SNN's weights with the corresponding pre-trained ANN weights and then train it further to fit the network on spiking inputs, using integrate-and-fire neurons and BPTT with a surrogate gradient. This was done to reduce the time to train each SNN from scratch which typically takes 100s of epochs and runs in days. It is to be noted that hybrid/transfer learning is a standard practice in deep SNNs [37], [43] as it enables faster convergence. Further, since we use the same weight initialization and hyper-parameter values as baseline SNN BPTT, *skipper* starts at an equal footing with the baseline.

Next, we used Poisson-based rate encoding to convert the CIFAR10 and CIFAR100 image datasets into spiking data. Unlike the CIFAR10/100 datasets that are recorded using a frame-based camera, the DVS Gesture Recognition, as well as the N-MNIST datasets, are already in the spiking format as they are recorded with a Dynamic Vision Sensor as sparse asynchronous binary address events (x, y, p, t), where (x, y) denote the spatial address, and (p, t) denote the polarity and time-stamp of each event stimulus respectively. The DVS Gesture dataset was recorded using the DVS-128 camera and consists of 11 hand gestures (such as clapping, waving, arm rotation etc.) performed by 29 different individuals under 3 illumination conditions and the problem is to classify an action sequence into an action category. Pre-processing of the dataset and ANN pre-training are performed using the techniques described in [44] and [45]. The N-MNIST dataset was created by performing saccadic movements of the Asynchronous Time-based Image Sensor (ATIS) [46] while being exposed to images from the MNIST [47] dataset which were displayed on an LCD screen.

For the timing and memory measurements, we supplied $\sim$ 40 – 100% of the training dataset samples in multiple mini-batches and averaged it over 20 such iterations after a warm start. The overall GPU memory consumption is measured using the 'pynvml' [48] tool and the details of library overheads and tensor memory consumption are obtained using the PyTorch `max_memory_allocated()` and `max_memory_reserved()` [49] APIs. The computation runtime is measured for each mini-batch and consists of the `forward()`, `backward()` and `weight_update()` calls on the network. Finally, all training and benchmarking is done on a server consisting of the NVIDIA A100 80GB GPUs [50] supported by Intel(R) Xeon(R) Silver 4314 @ 2.40GHz CPUs, using Pytorch version 1.10 and CUDA version 11.2. Following are our detailed evaluation results.

A. SNN training memory cost vs #checkpoints C

Figures 7 (a), (b) , (c) and (d) show the overall peak GPU memory consumption during SNN training as a function of the number of checkpoints C for a constant batch size B and timesteps T. We notice that the memory consumption reduces with more checkpoints and reaches a minimum value, after which it starts to increase. For example, in the case of VGG5 simulated for $T = 100$ timesteps, the memory consumption is the lowest when $C = \sqrt{T}$, as explained in Section V-A. For small values of C, the memory consumption of each time-segment dominates the overall activation memory, whereas for large Cs, it is dominated by the memory required for storing intermediate states in between checkpoints (as per equation 3). Further, the computational overhead of checkpointing is shown on the right axis of Figures 7(a), (b), (c) and (d) as a function of C. Compared to the baseline SNN, the checkpointed version incurs around 30% overhead, on average, for any number of checkpoints as long as the forward pass is re-computed once. We observe these trends for the all the 4 networks.

The key takeaway of this study is that, following a 30% increase in training time of checkpointed SNNs compared to the baseline, the run-time remains more or less constant with increasing values of C, and the memory consumption is minimum at $C = \sqrt{T}$.

B. *SNN accuracy with* Skipper

Table I shows the network accuracy on the testing set for 5 different networks of varying sizes and depths with 4 different datasets. The accuracy of the checkpointed networks is reported to verify the correctness of our implementation, and its slight variation from the baseline accuracy is due to the

Training Details	VGG5	VGG11	ResNet20	LeNet	custom-Net
Baseline (BPTT)	0.8734	0.6623	0.8716	0.8897	0.9681
Checkpointed	0.8714 (C=4)	0.6648 (C=5)	0.8704 (C=5)	0.8889 (C=10)	0.9696 (C=4)
Skipper	0.8744 (p=70)	0.6648 (p=50)	0.8728 (p=52)	0.8933 (p=70)	0.9635 (p=70)
Trunc. BPTT	0.8715 (trW=25)	0.5673 (trW=25)	0.8593 (trW=50)	0.8882 (trW=40)	-
# layers	conv(3)+lin(3)	conv(9)+lin(3)	conv(20)+lin(1)	conv(5)+lin(1)	conv(3)+lin(1)
Dataset	CIFAR10	CIFAR100	CIFAR10	DVS-Gesture	N-MNIST
Timesteps (T)	100	125	250	400	300
Batch Size	128	128	128	32	256

Table I: SNN test accuracy of VGG5, VGG11, ResNet20, LeNet and custom-Net with four different training techniques (BPTT, Checkpointed, Skipper, and TBPTT). C: #checkpoints, p: percentile of skipped timesteps, trW: truncation window for TBPTT, $conv$: convolution layers, and lin: linear layers for a particular network.

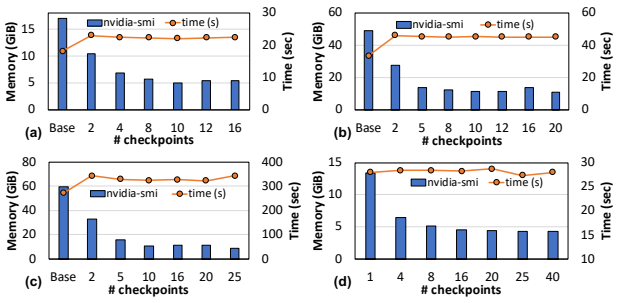
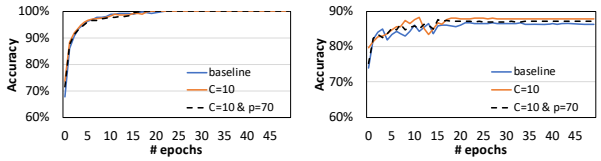


Figure 7: Overall peak GPU memory consumption and computation time (for a fixed number of iterations) vs number of checkpoints for (a) VGG5+CIFAR10, T=100, B=128, (b) VGG11+CIFAR100, T=125, B=128 (c) ResNet20+CIFAR10, T=250, B=128, (d) LeNet+DVS-getsure, T=400, B=32.



(a) Training **(b)** Validation

Figure 8: Accuracy vs #epochs for training LeNet SNN on DVS-gesture from scratch. B=64, T=400.

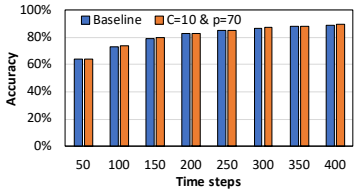


Figure 9: Accuracy vs #timesteps for LeNet SNN on DVS-gesture @ B=32, T=400 trained with baseline and *skipper*.

stochasticity of the spiking input, dropout patterns, and optimizer. We notice that the SNN accuracy with *skipper* is quite competitive, if not slightly better, than the baseline accuracy. We surmise that this is due to an additional regularization that the network receives in the temporal domain, which is akin to a spatial dropout in conventional DNNs, that enables the network to generalize better [51]. However, unlike spatial dropouts, our temporal dropout takes place at a much higher granularity i.e., we *completely* skip the computations of all the layers in a given timestep, and the timesteps at which to drop computations are not chosen randomly, but are based on a well-defined heuristic. Further, we report the accuracy for the maximum fraction of timesteps that we were able to skip without losing any performance and demonstrate the skipping of up to 70% of the timesteps with little accuracy loss.

Note that, for networks with a higher $\frac{T}{L_n}$ ratio, we can skip more timesteps (refer to Equation 7), e.g., VGG5+CIFAR10 has a higher $\frac{T}{L_n}$ ratio than ResNet20+CIFAR10 and therefore, has a higher time-skipping fraction. Additionally, the dataset complexity also dictates the number of timesteps that can be skipped (e.g., VGG11 on CIFAR100).

Finally, in addition to frame-based datasets, our proposed technique is also effective on event-based data as shown by the accuracy of LeNet on DVS gesture and that of a custom network on N-MNIST. The SNNs were trained from scratch for 50 and 100 epochs respectively, and for both datasets, *skipper* achieved a similar convergence to baseline and checkpointed schemes. As proof of concept, Figure 8 reports the accuracy curves for training an SNN from scratch for the baseline, checkpointed and *skipper* regimes, and Figure 9 plots the accuracy against #timesteps for the baseline and *skipper* techniques, thus further establishing *skipper*'s competitiveness with the other techniques.

C. *SNN training computation cost vs batch size*

Next, we analyze the impact of batch size on training time. Figures 10(a), (b), (c) and (d) report the computational overhead of training SNNs with plain checkpointing and *skipper* strategies as a function of batch size, over baseline.

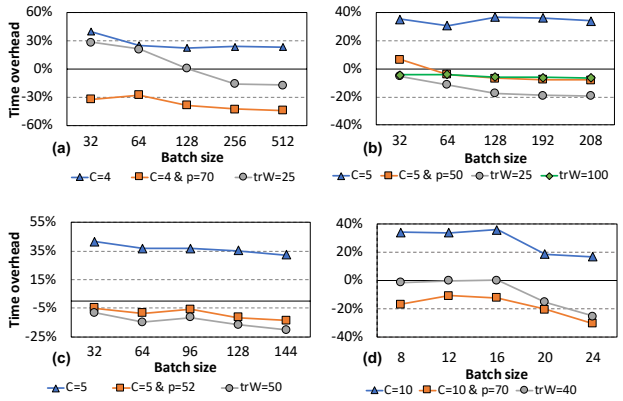


Figure 10: Computational overhead of checkpointing (C), *skipper* (C,p), and TBPTT (trW) training regimes vs batch size for (a) VGG5+CIFAR10, T=100; (b) VGG11+CIFAR100, T=125; (c) ResNet20+CIFAR10, T=250; and (d) LeNet+DVS-gesture, T=400.

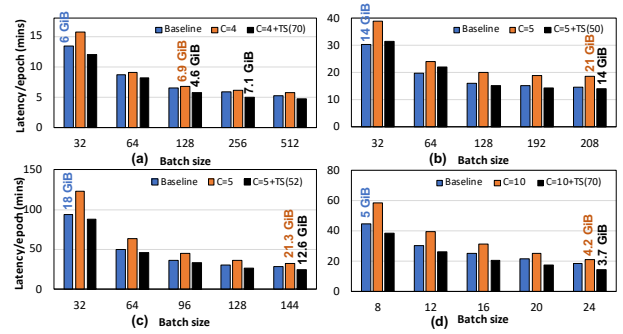


Figure 11: SNN end-to-end training latency vs batch size for (a) VGG5+CIFAR10, (b) VGG11+CIFAR100, (c) ResNet20+CIFAR10, and (d) LeNet+DVS-gesture. Number above a bar reports the corresponding memory consumption.

For VGG5 with CIFAR10 in Figure 10(a), the computational cost is measured using the entire training dataset for all training strategies. The checkpointed VGG5 network incurs a run-time overhead of 20% – 40% compared to baseline BPTT, and this overhead is lower for larger batch sizes. For the *skipper* training regime, this computational overhead is not only amortized but completely alleviated, leading to a 30% – 40% *reduction* in training time compared to the baseline. This can be attributed to a large number of computations of intermediate states that are dropped (∼70%) during the second forward pass of checkpointing.

In VGG11 with CIFAR100 (Figure 10(b)), there is a similar reduction in computational cost of the network trained with *skipper* compared to its plain checkpointed counterpart (ranging from 29% to 42%), which also increases with the batch size. However, for small batch sizes (B=32), the *skipper* network still incurs a small overhead of 6%

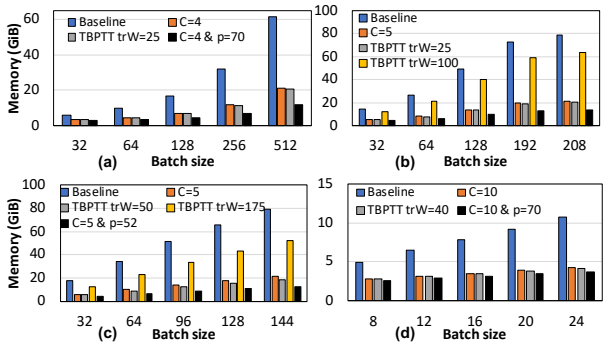


Figure 12: Overall GPU memory consumption of baseline BPTT, checkpointing, *skipper* and TBPTT techniques vs batch size for (a) VGG5+CIFAR10, T=100, (b) VGG11+CIFAR100, T=125, (c) ResNet20+CIFAR10, T=250, (d)LeNet+DVS-gesture, T=400.

compared to the baseline. For larger batch sizes (from B=64 onwards), *skipper* reduces the computational cost by ∼4% to ∼8% compared to baseline-BPTT. We notice that the computational cost savings in this network are lower than those obtained in VGG5. This can be attributed to the lower time-skipping ratio (50% in VGG11 compared to 70% in VGG5), which eventually depends on the $\frac{T}{L_n}$ ratio of each network. The $\frac{T}{L_n}$ ratio is higher for VGG5 than VGG11, which allows it to skip more time-steps (refer to eq. 7) without potentially losing much accuracy.

Similar conclusions can be drawn for ResNet20 on CIFAR10 (Figure 10(c)), in which the *skipper* technique saves ∼5% to ∼14% of the computation time compared to the baseline SNN as its $\frac{T}{L_n}$ ratio lies in between that of VGG5 and VGG11. For LeNet on DVS-gesture dataset, the computation time is 18% to 30% lower than the baseline (Figure 10(d)). Finally, in the case of N-MNIST dataset, *skipper* (C=4, p=70) reduces baseline-BPTT computation time by 8%.

To summarize, *skipper* successfully alleviates the computational overhead of plain checkpointing as is demonstrated on networks and time steps of varying sizes, with savings as high as 40% compared to the baseline SNN training time.

D. SNN end-to-end training latency vs batch size

Figures 11 (a), (b), (c) and (d) report the end-to-end training latency in minutes of the different strategies at a constant T against batch size for a single training epoch. In each of the cases across the 4 networks, given a constant memory budget, *skipper* can enable training of larger batches with reduced training latency. E.g., in the case of VGG11+CIFAR100 shown in Figure 11 (b), for a memory budget sufficient to fit a baseline-BPTT training with mini-batch of size 32 (memory consumption reported on top of bar), *skipper* (C=5, p=50) can enable training with batch size of up to 208, leading to 52% reduction in training

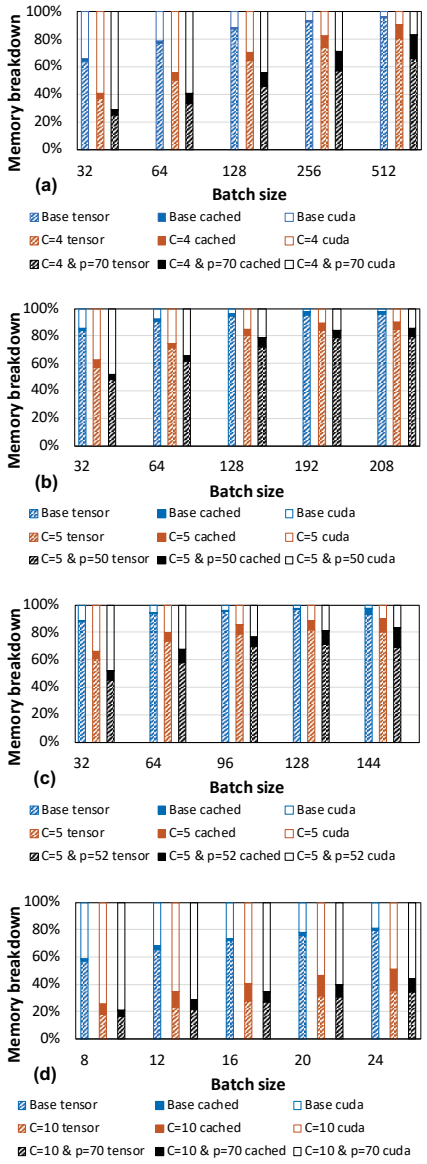


Figure 13: Breakdown of overall GPU memory consumption of baseline (Base), checkpointing (C), and *skipper* (C,p) techniques vs batch size for (a) VGG5 + CIFAR10, T=100, (b) VGG11 + CIFAR100, T=125, (c) ResNet20 + CIFAR10, T=250, (d) LeNet + DVS-gesture, T=400.

latency. A similar trend can be observed for the other networks evaluated in this study. Note that the largest batch size used for our batch-sweep experiments is set based on the maximum memory consumption of baseline BPTT supported by the underlying GPU.

E. SNN training memory cost vs batch size

Figures 12(a), (b), (c) and (d) report the overall memory consumption for different training strategies of VGG5 on CIFAR10, VGG11 on CIFAR100, ResNet20 on CIFAR10 and

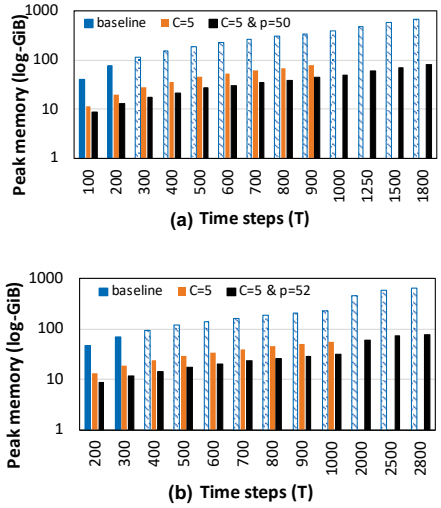


Figure 14: GPU memory consumption of baseline, checkpointing and *skipper* techniques vs time steps for (a) VGG11+CIFAR100, (b) ResNet20+CIFAR10. Patterned bars show extrapolated memory cost of baseline BPTT (overflows 80GB GPU DRAM).

LeNet on DVS-gesture SNNs, respectively. As is apparent from the figures, checkpointing helps to reduce the overall memory consumption of training by $1.7\times-3\times$ for VGG5, by $2.6\times-3.7\times$ for VGG11, by $3\times-3.7\times$ for ResNet20, and by $2\times-2.5\times$ for LeNet. As expected, the savings are higher with larger batch sizes. Further, for the N-MNIST dataset, the overall training memory footprint of the checkpointed SNN (C=4) is $1.6\times$ lower and that of *skipper* (C=4, p=70) is $2\times$ lower compared to the baseline.

The final memory consumption depends on several factors within the framework runtime such as its memory allocation policy, tensor alignment and CUDA context [52]. In order to understand the impact of these factors, we plot a breakdown of the overall memory consumption in terms of the space occupied by tensors, and the overheads of the deep learning framework and the CUDA context in Figures 13(a), (b), (c) and (d). In the figures, for each of the batch sizes, we plot three bars - one for the base configuration, one for the checkpointing scheme, and one for *skipper*. (e.g., in Figure 13(a), the three bars correspond to baseline BPTT, checkpointing (C=4), and *skipper* (C=4 & p=70) configurations). Each bar in turn reports the proportion of memory allocated to tensors, PyTorch cache and CUDA context. For small networks and batch sizes, the CUDA context overhead can be quite large (ranging from 50% to 80% for smallest batch size in the time-skipped networks). This means that the actual memory savings are *higher* than those shown by the overall figures (Figure 12). Thus, if we only consider the tensor memory consumption and discount the CUDA context and PyTorch's caching overheads, the memory savings with checkpointing are higher still, in the range of $3\times-3.4\times$

for VGG5, 3.8×−4.2× for VGG11, ∼4× for ResNet20 and ∼5.8× for LeNet. Note that the amount of memory savings depends on a number of factors such as the network depth (L_n), timesteps (T), batch-size (B) and the number of checkpoints (C). For this experiment, we have chosen C to be a factor of T for each network, so that the length of every checkpointed time-segment T/C in memory is the same.

Moreover, *skipper* helps to reduce the memory consumption even further compared to plain checkpointing, and the savings come as a by-product of our strategy to reduce the computational overhead (i.e. due to the formation of a shallower computation graph in the second forward pass). Quantitatively, it lowers the overall memory cost by another 1.2× - 1.7× for VGG5, by 1.2× - 1.5× for VGG11 and by 1.4× - 1.7× for ResNet20, compared to plain checkpointing. When considering tensors alone, the memory savings are 1.8× - 2.1× for VGG5, 1.5× - 1.6× for VGG11 and ∼2× for ResNet20, compared to plain checkpointing with C=5. For N-MNIST, the corresponding memory savings are 2.6× and 3.35× with plain checkpointing and *skipper* respectively.

In summary, the memory savings of the checkpointed SNNs range from 3.4× (2.6×) to 8.4× (4.3×) and on average 6.7× (4.2×) with (without) time-skipping, as compared to baseline BPTT for the networks studied.

F. Comparison with TBPTT

Next, we compare our proposed training approach with TBPTT [27] on all three metrics viz. memory consumption, run-time and accuracy. For VGG5 SNN, Figure 12 (a) shows the overall memory consumption of TBPTT with a truncation window of 25 (TBPTT trW=25), where the truncation window is chosen for an iso-memory comparison with the checkpointing approach. Table I shows the corresponding test accuracy of this network, which is comparable to the baseline, plain checkpointing and *skipper* techniques. The computational overhead, on the other hand, is higher than *skipper*. Thus, for similar accuracy, *skipper* performs better in terms of both memory and computational cost than TBPTT trW=25. In the case of VGG11 SNN, a truncation window of 25 has a similar memory consumption to its checkpointed variant. At this level of temporal unrolling, the network accuracy of TBPTT trW=25 is ∼57% as reported in Table I, which is 9% lower than the baseline and other techniques. Figure 10(b) shows the computational savings of TBPTT trW=25 and trW=100, both of which perform as well as, if not better than *skipper*. Although its computational savings are higher than *skipper*, it comes at a huge cost to network accuracy. TBPTT trW=100 improves the network accuracy by a meager 1%, and also simultaneously foregoes the memory advantage (refer Figure 12(b) – TBPTT trW=100).

For ResNet20 SNN on CIFAR10, the network accuracy of TBPTT trW=50 is ∼1% lower than the baseline (Table I), with a similar memory consumption to checkpointing (Figure 12(c)) and higher computational savings compared to *skipper* (Figure 10(c)). However with a larger truncation window (trW=175) the accuracy does not improve much (85.94%) and the memory savings are also lost. Finally, for LeNet with DVS inputs, the accuracy is competitive at similar memory cost to checkpointing (trW=40, Table I), and memory savings of *skipper* are limited (due to small network and batch sizes). However, the corresponding time savings are less for the TBPTT approach compared to *skipper* (Figure 10 (d)). This is due to a small truncation window compared to T that leads to more number of backward passes.

The key takeaway of these experiments is that for a memory cost similar to activation checkpointing, TBPTT is quite effective in reducing the computational cost of training SNNs for relatively large time windows, but fails to provide competitive performance on deeper networks compared to all the other approaches presented.

G. SNN training memory cost vs timesteps

Figure 14(a) reports the overall peak GPU memory consumption on a log scale as a function of the computation time-steps for training VGG11 with CIFAR100 in the baseline, checkpointed (C=5) and *skipper* (C=5 & p=50) regimes. We can see a linear increase in the memory consumption for the baseline technique, due to which the GPU runs out of memory as T is increased from 200 to 300 (the patterned blue bars show extrapolated memory requirements, thus incapable of running in the available 80GB memory budget of A100 GPU) and training fails. The checkpointed network memory growth is sub-linear and allows training of networks with up to 4.5 × as many time-steps ($T = 900$). The growth in memory consumption of the *skipper*-trained network is even slower and is, therefore, able to scale to nearly twice as many timesteps as plain checkpointing i.e. up to $T = 1800$. This is because the memory footprint of time-skipped SNN is nearly half of the checkpointed network. Thus, for a constant memory budget, the plain checkpointed network and the time-skipped network can scale to 4.5× and 9× longer timesteps respectively, than the maximum timesteps ($T = 200$) supported for the baseline SNN for VGG11. Similar trends are observed for ResNet20 SNN as well (see Figure 14(b)), where the checkpointed and *skipper* networks scale to 3.3× and 9.3× respectively, compared to the maximum for baseline SNN (at $T = 300$).

These experiments show the scalability of *skipper* on top of the plain checkpointing technique and its advantage in running large models such as VGG11 or ResNet20 for longer time steps for improving accuracy.

H. Experiments on an edge device

Next, we study the efficacy of our approach by running an SNN on a 5W NVIDIA Jetson Nano [53] edge computing

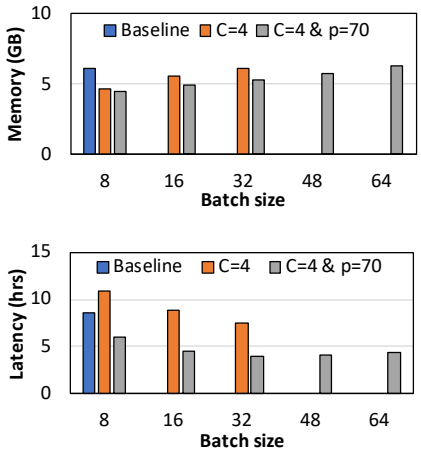


Figure 15: VGG5+CIFAR10, T=100, (a) memory consumption and (b) latency per epoch vs batch size on NVIDIA Jetson Nano.

	Config.	Accuracy	GiB
TBPTT-LBP [28], T=20	trW=10	84.46%	6.15
	trW=20	84.25%	9.27
This work, T=20	C=2	84.59%	6.26
	C=2 & p=20	84.07%	6.16

Table II: Comparison of checkpointing and *skipper* against TBPTT-LBP [28] on AlexNet+CIFAR10, B=256.

device as shown in Figure 15. The board has 4GB of unified memory that is shared by both the CPU and the GPU. Since the CUDA context takes up a large proportion of the memory (∼2GB), we allocated another 4GB as swap space, to be able to demonstrate the different schemes. Figure 15(a) shows the overall memory consumption vs batch size and Figure 15(b) shows the corresponding training latencies for the baseline, checkpointing (C=4) and *skipper* (C=4 & p=70) schemes while training VGG5 SNN on CIFAR10. As can be seen from the figures, the jetson board could not support the baseline SNN training beyond a batch size of 8. The checkpointed SNN was able to run with a batch size of up to 32, whereas the corresponding *skipper* version enabled a batch size of 64. As a result, the corresponding training latency of *skipper* at B=64 was reduced by 50% compared to the baseline, for a similar memory footprint. Thus, our proposed techniques can directly lead to power and energy savings due to a smaller memory footprint and reduced computational complexity on the edge device.

I. Comparison with [28]

In this section, we quantitatively compare our proposed approach against a recent proposal [28], which uses a combination of temporal truncation (TBPTT) and local backpropagation (LBP) [14] to reduce the SNN training memory consumption. As already noted in the previous sections, temporal truncation helps to reduce the activation memory

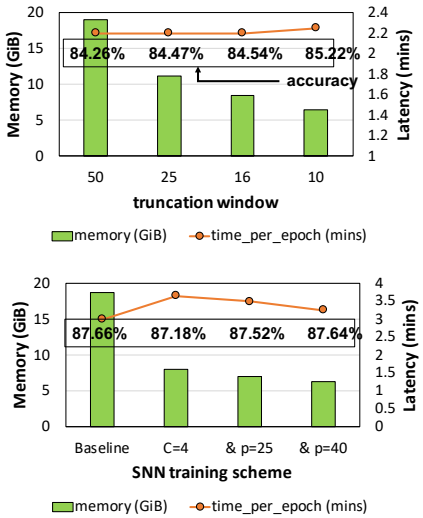


Figure 16: Comparison of memory/time/accuracy vs (a) truncation window of TBPTT-LBP [28] and (b) proposed approaches on AlexNet+CIFAR10, T=50, B=256.

consumption of the SNN, but adversely affects the network accuracy. Ref. [28] proposes the use of local classifiers attached to some of the SNN layers in combination with temporal truncation to help alleviate the over-fitting issue. The local classifiers, however, incur an additional memory cost, albeit small, due to their own weights. For comparison, we train AlexNet SNN (topology details in [28]) with CIFAR10 on both frameworks with 20 and 50 timesteps. As per their best-performing configuration, the local classifiers are attached at layers 4 and 8 and the network is trained for 100 epochs with the settings mentioned in the paper.

Table II reports the TBPTT-LBP and checkpointing/*skipper* accuracy and memory cost at T=20. In the former, enlarging the truncation window (the number of unrolled timesteps) from 10 to 20 increases the memory consumption, but does not help to improve the network accuracy which remains at ∼84%. Notably, the checkpointing (C=2) and *skipper* (C=2 & p=20) techniques achieve a similar accuracy at similar or lower memory cost compared to TBPTT-LBP. Further, Figure 16(a) reports the memory cost, the latency per epoch and the accuracy of the TBPTT-LBP [28] algorithm for training AlexNet on CIFAR10 as a function of the truncation window at T=50. Compared with T=20, the TBPTT-LBP accuracy does not improve much at a higher number of timesteps (T=50), even for larger truncation windows that increase the memory cost of training. In contrast, in the proposed framework (Figure 16 (b)), the SNN accuracy improves with more timesteps (by ∼3%) and *skipper* maintains this accuracy with up to 40% of skipped timesteps (C=4, p=40). The corresponding memory consumption of our proposed schemes is also similar or lower than TBPTT-LBP.

In summary, TBPTT-LBP achieves reasonable accu-

racy and low memory footprint when trained with fewer timesteps, but its accuracy does not scale with an increased number of timesteps. Checkpointing and *skipper* on the other hand, not only achieve improved accuracies with larger timesteps but do so at lower memory costs.

VIII. Discussion

A. Challenges and uniqueness of SNN

Although the re-computation and time-skipping approaches proposed in this work are also applicable to DNNs, they have been independently demonstrated in prior literature in separate contexts. Specifically, re-computation has been utilized to reduce the training memory footprint of DNNs, whereas dropout has been used as a regularization technique to improve the network's generalization ability and also in some cases, to lower its computational footprint. However, combining these two techniques in the DNN context is not feasible, as that would necessitate the dropping out of an entire layer during the second forward pass to get reasonable latency reduction, which will obstruct the flow of information across layers, affecting accuracy. Activation checkpointing with time-skipping, however, fits quite naturally with the SNN paradigm, as we recompute and skip timesteps in the *temporal* dimension without hindering information flow across layers. Further, since *skipper* monitors the time-dependent SNN activity in the form of spike-sums to determine which timesteps to skip, it utilizes the specific temporal dynamics that are unique to SNNs. Although it is possible to skip timesteps in the RNN context, SNNs pose the additional challenge of larger number of timesteps and spike-based discretization, due to which their real gradients cannot be calculated. Remarkably, *skipper* efficiently navigates these challenges and yields comparable accuracy, while reducing the memory and computational cost of SNN training.

B. Training SNNs on neuromorphic accelerators (TrueNorth, Loihi)

The literature is replete with designs for neuromorphic chips out of which TrueNorth [7] and Loihi/Loihi2 [2] are fully functional prototypes that can support SNNs. However, our proposed activation checkpointing and *skipper* approaches aim to optimize offline SNN training for which GPUs are more suitable as they have the software infrastructure to support arbitrary functions and control statements. On the other hand, chips such as TrueNorth and Loihi are more suitable for low-power deployment scenarios after the SNN has been trained. Specifically, TrueNorth's circuits are optimized for inference and do not support plasticity. Further, although Loihi supports learning on-chip, it does so in an incremental fashion, with a limited set of learning rules and a batch size of 1 due to several architectural constraints designed to keep the circuits low power. As a result, it is not suitable for batched training of SNNs. However, SNNs trained offline using our approach can be deployed on TrueNorth/Loihi [54], [55] or any other neuromorphic platform for online learning or inference.

IX. Conclusion

In this work, we have explored a recomputation-based technique viz. activation checkpointing, to alleviate the large memory cost associated with SNN training. Although checkpointing decreases memory consumption by $3\times-4.3\times$, it still incurs a significant computation overhead ($\sim$33%) leading to longer training time. To mitigate this overhead, we proposed a novel technique, called *Skipper* (checkpointing-with-timeskipping), that approximates the SNN-BPTT algorithm to skip low-activity time steps and thus, reducing the computational overhead of plain checkpointing by 29% to 70%. This translates to a training time reduction of 4% to 40% compared to the baseline BPTT with little to no loss of accuracy. The time-skipped network also reduced the memory consumption even further and yielded up to $8.4\times$ lower memory cost. Our techniques were thoroughly evaluated on state-of-the-art networks and datasets including a neuromorphic vision-based dataset to demonstrate the applicability of the technique to other neuromorphic architectures. In addition to easing SNN training deployment under resource-constrained environment, we believe our research will be beneficial to the wider neuromorphic community in general towards exploring larger and deeper SNN models by enabling memory and compute efficient training. Our PyTorch implementation can be accessed at https://github.com/sms821/Training-SNN-with-checkpointing-and-time-skipping-in-PyTorch.

Acknowledgment

This research is supported in part by the National Science Foundation grant #1955815, #1763681 and the Semiconductor Research Corporation-CBRIC center. We thank the funding agencies for their support. Further, we thank the authors of [28] for sharing their code-base with us. We also thank Sadia Anjum Tumpa for her timely help and feedback.

References

[1] E. Chicca, F. Stefanini, and G. Indiveri, "Neuromorphic electronic circuits for building autonomous cognitive systems," *Proceedings of IEEE*, 2013.

[2] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, P. Joshi, A. Lines, A. Wild, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. PP, no. 99, pp. 1–1, 2018.

[3] C. Pehle, S. Billaudelle, B. Cramer, J. Kaiser, K. Schreiber, Y. Stradmann, J. Weis, A. Leibfried, E. Müller, and J. Schemmel, "The brainscales-2 accelerated neuromorphic system with hybrid plasticity," *arXiv preprint arXiv:2201.11063*, 2022.

[4] S.-C. Liu and T. Delbruck, "Neuromorphic sensory systems," *Current Opinion in Neurobiology*, vol. 20, no. 3, pp. 288–295, 2010.

[5] G. Gallego, T. Delbruck, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis *et al.*, "Event-based vision: A survey," *arXiv preprint arXiv:1904.08405*, 2019.

[6] M. Davies, A. Wild, G. Orchard, Y. Sandamirskaya, G. A. F. Guerra, P. Joshi, P. Plank, and S. R. Risbud, "Advancing neuromorphic computing with loihi: A survey of results and outlook," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 911–934, 2021.

[7] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[8] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, 2006.

[9] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, Nov 2019. [Online]. Available: https://ieeexplore.ieee.org/ielaam/79/8887548/8891809-aam.pdf

[10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[12] P. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[13] N. Perez-Nieves and D. F. Goodman, "Sparse spiking gradient descent," *arXiv preprint arXiv:2105.08810*, 2021.

[14] J. Kaiser, H. Mostafa, and E. Neftci, "Synaptic plasticity dynamics for deep continuous local learning (decolle)," *Frontiers in Neuroscience*, vol. 14, p. 424, 2020. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2020.00424

[15] F. Zenke and S. Ganguli, "Superspike: Supervised learning in multilayer spiking neural networks," *Neural computation*, vol. 30, no. 6, pp. 1514–1541, 2018.

[16] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein, and W. Maass, "A solution to the learning dilemma for recurrent networks of spiking neurons," *Nature communications*, vol. 11, no. 1, pp. 1–15, 2020.

[17] E. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines," *Frontiers in Neuroscience*, vol. 11, p. 324, Jun 2017. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2017.00324/full

[18] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International joint conference on neural networks (IJCNN)*. ieee, 2015, pp. 1–8.

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

[20] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in Neuroscience*, vol. 11, 2017. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2017.00682

[21] B. Nessler, M. Pfeiffer, and W. Maass, "STDP enables spiking neurons to detect hidden causes of their inputs," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 1357–1365.

[22] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "Stdp-based spiking deep convolutional neural networks for object recognition," *Neural Networks*, 2017.

[23] F. Zenke and E. O. Neftci, "Brain-inspired learning on neuromorphic substrates," *Proceedings of the IEEE*, pp. 1–16, 2021.

[24] J. Menick, E. Elsen, U. Evci, S. Osindero, K. Simonyan, and A. Graves, "A practical sparse approximation for real time recurrent learning," *arXiv preprint arXiv:2006.07232*, 2020.

[25] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016. [Online]. Available: https://arxiv.org/abs/1604.06174

[26] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[27] I. Sutskever, *Training recurrent neural networks*. University of Toronto Toronto, ON, Canada, 2013.

[28] W. Guo, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Efficient training of spiking neural networks with temporally-truncated local backpropagation through time," 2022. [Online]. Available: https://arxiv.org/abs/2201.07210

[29] S. Singh, A. Sarma, N. Jao, A. Pattnaik, S. Lu, K. Yang, A. Sengupta, V. Narayanan, and C. R. Das, "Nebula: A neuromorphic spin-based ultra-low power architecture for snns and anns," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 363–376.

[30] D. Lee, G. Lee, D. Kwon, S. Lee, Y. Kim, and J. Kim, "Flexon: A flexible digital neuron for efficient spiking neural network simulations," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 275–288. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00032

[31] S. Narayanan, K. Taht, R. Balasubramonian, E. Giacomin, and P.-E. Gaillardon, "Spinalflow: An architecture and dataflow tailored for spiking neural networks," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 349–362.

[32] J.-J. Lee, W. Zhang, and P. Li, "Parallel time batching: Systolic-array acceleration of sparse spiking neural computation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 317–330.

[33] B. Dauvergne and L. Hascoët, "The data-flow equations of checkpointing in reverse automatic differentiation," in *Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 566–573.

[34] M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama, "A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

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

[36] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-scale Image Recognition," in *ArXiv*, 2014.

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

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645.

[39] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 and cifar-100 datasets," "https://www.cs.toronto.edu/~kriz/cifar.html".

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[41] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza *et al.*, "A low power, fully event-based gesture recognition system," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7243–7252.

[42] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, nov 2015.

[43] S. Lu and A. Sengupta, "Neuroevolution guided hybrid spiking neural network training," *Frontiers in Neuroscience*, vol. 16, 2022.

[44] B. Rückauer, N. Känzig, S.-C. Liu, T. Delbrück, and Y. Sandamirskaya, "Closing the accuracy gap in an event-based visual recognition task," *ArXiv*, vol. abs/1906.08859, 2019.

[45] S. Singh, A. Sarma, S. Lu, A. Sengupta, V. Narayanan, and C. R. Das, "Gesture-snn: Co-optimizing accuracy, latency and energy of snns for neuromorphic vision sensors," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '21. IEEE Press, 2021. [Online]. Available: https://doi.org/10.1109/ISLPED52811.2021.9502506

[46] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 259–275, jan. 2011.

[47] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[48] "Python bindings to the nvidia management library," https://github.com/gpuopenanalytics/pynvml.

[49] "Memory management," https://pytorch.org/docs/stable/notes/cuda.html#cuda-memory-management.

[50] "NVIDIA A100 TENSOR CORE GPU," https://www.nvidia.com/en-us/data-center/a100/.

[51] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[52] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, *Estimating GPU Memory Consumption of Deep Learning Models*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1342–1352. [Online]. Available: https://doi.org/10.1145/3368089.3417050

[53] "Jetson nano," "https://developer.nvidia.com/embedded/jetson-nano".

[54] K. Stewart, G. Orchard, S. B. Shrestha, and E. Neftci, "On-chip few-shot learning with surrogate gradient descent on a neuromorphic processor," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Sep. 2020, pp. 223–227. [Online]. Available: http://arxiv.org/pdf/1910.04972

[55] R. Massa, A. Marchisio, M. Martina, and M. Shafique, "An efficient spiking neural network for recognizing gestures with a dvs camera on the loihi neuromorphic processor," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–9.