UNIVERSITY OF CALIFORNIA SAN DIEGO

Reshaping Deep Neural Networks for Efficient Hardware Inference

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

in

Computer Science (Computer Engineering)

by

Alireza Khodamoradi

Committee in charge:

Professor Ryan C. Kastner, Chair
Professor Gert Cauwenberghs
Professor Sicun Gao
Professor Steven Swanson
Professor Jishen Zhao

2021

The Dissertation of Alireza Khodamoradi is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

For my mother, Azar.

EPIGRAPH

Entities are not to be multiplied beyond necessity.

William of Ockham

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Ryan Kastner for his support as the chair of my committee. I would also like to acknowledge Kristof Denolf, without whom my research would have no doubt taken longer.

Chapter 2, in full, is a reprint of the material as it appears in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays 2021. Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in IEEE Transactions on Emerging Topics in Computing 2017. Alireza Khodamoradi and Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part is currently being prepared for submission for publication of the material. Alireza Khodamoradi, Olivia Weng, Nojan Sheybani, Kristof Denolf, Farinaz Koushanfar, and Ryan Kastner. The dissertation author was the primary investigator and author of this material.

Chapter 5, in full, is a reprint of the material as it appears in the International Joint Conference on Neural Networks 2021. Alireza Khodamoradi, Kristof Denolf, Kees Vissers, and Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

VITA

2002	B.S. in Electrical Engineering, Azad University of Najafabad, Iran
2008	M.S. in Electrical Engineering, University of Kerman, Iran
2015	M.A.S in Wireless Embedded Systems, University of California San Diego
2021	Ph.D. in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Alireza Khodamoradi, Kristof Denolf, Ryan Kastner, "S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks", *FPGA* 2021

Stephen Neuendorffer, Alireza Khodamoradi, Kristof Denolf, Abishek K. Jain, Samuel Bayliss, "The Evolution of Domain-Specific Computing for Deep Learning", *IEEE Circuits and Systems* 2021

Alireza Khodamoradi, Kristof Denolf, Ryan Kastner, Kees Vissers, "ASLR: an Adaptive Scheduler for Learning Rate", *IJCNN* 2021

Stephen Tridgell, David Boland, Philip H.W. Leong, Alireza Khodamoradi, Ryan Kastner, Siddhartha, "Real-time Automatic Modulation Classification using RFSoC", *RAW* 2020

Murad Qasaimeh, Kristof Denolf, Alireza Khodamoradi, Lisa Halder, Michaela Blott, Jack Lo, Kees Vissers, Joseph Zambreno, Phillip H.Jone, "Benchmarking Vision Kernels and Neural Network Inference Accelerators on Embedded Platforms", *JSA* 2019

Alireza Khodamoradi, Ryan Kastner, "O(N)-Space Spatiotemporal Filters for Reducing Noise in Neuromorphic Vision Sensors", *IEEE Transactions on Emerging Topics in Computing (TETC)*, 2017

Andrew Lanez, Sachin B. Sundramurthy, Alireza Khodamoradi, "RFNoC & Vivado HLS Challenge - Team Rabbit Ears: ATSC Receiver", *Proceedings of the 7th GNU Radio Conference (GRCon)*, 2017

Alican Nalci, Alireza Khodamoradi, Ozgur Balkan, Fatta Nahab, Harinath Garudadri, "A computer vision based candidate for functional balance test", *Engineering in Medicine and Biology Society (EMBC)*, 2015

ABSTRACT OF THE DISSERTATION

Reshaping Deep Neural Networks for Efficient Hardware Inference

by

Alireza Khodamoradi

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2021

Professor Ryan C. Kastner, Chair

The latest Deep Learning (DL) methods for designing Deep Neural Networks (DNN) have significantly expanded our ability to train data processing systems. Coupled with exponential growth in available digital data, we have seen dramatic accuracy improvements in DNNs and widespread adoption of these models in different applications.

This increased demand has motivated innovations in DNN architecture design to deliver high-quality output. For example, advanced DL models can include irregular connections between their layers, have more parameters, and employ computationally complex neurons. Unfortunately, these new architectural additions often increase the implementation complexity of the DNNs on hardware, particularly when deploying DL models for inference in scale-out and

power-limited systems.

Currently, to deploy a DNN on a custom platform, an abstract of the DL model is used to create a functionally identical realization. However, because altering this abstract changes the functionality of the DL model, hardware designers keep the model unchanged for a lossless implementation.

This thesis shows that a co-design approach can improve the hardware implementation of DL models. In a co-design approach, the designer reshapes the DNN architecture to better fit a target processing platform and preserves its accuracy by retraining the model.

We describe a custom accelerator for Spiking Neural Networks (SNN) with improved computational cost and memory utilization because of reshaping the layers and neurons of the model. We then apply these changes to the existing SNN models and show that they can maintain their accuracy after the reshaping and retraining. In addition, we introduce novel applications for SNNs based on the new architecture. We also present a stochastic noise filter for pre-processing SSN's input with improved accuracy and memory utilization. Furthermore, we explain a reshaping method for Residual Networks (ResNet) to reduce their memory footprint while preserving their accuracy.

This thesis also introduces a method for accelerating the co-design process. Reshaping DL models can increase the complexity of their training stage. We present an auto tuner for the learning rate (an essential parameter for training DNNs) that simplifies the manual tuning for this parameter and can accelerate the retraining of DL models.

Chapter 1

Introduction

Deep Learning (DL) models are a subset of Artificial Neural Networks (ANN). These data processing systems learn how to process their inputs to produce desirable outputs similar to biological neural networks (e.g., mammal's brain). Recent advancements in DL methods have shown significant improvements in training these models and have revealed their capabilities in producing high-quality products in different industries, including self-driving cars, language translation, healthcare, virtual assistants, and many others.

Generally, a DNN includes several layers, and each layer extracts different levels of features from the input. For example, in a model designed for image classification, earlier layers extract edges or corners, and later layers pull the number of objects in the image or determine if the input is an image of a dog or a duck.

Usually, each layer passes its output forward to the next layer. However, neural networks can also include irregular connections to allow their layers to pass their output to non-adjacent layers. These irregular connections are essential for training specific models and applications. For example, in Residual Networks (ResNet) [38], skip connections increase the network accuracy and make it possible to train deeper models with more layers, or in UNets [93], crop-and-copy connections are vital for high-quality image segmentation.

In a DL model, a layer is a collection of neurons. Each neuron has input connections from presynaptic neurons and output connections to postsynaptic neurons. Furthermore, each

connection has a strength represented by a scalar, viz. weight. Finally, a non-linear function governs the relationship between the neuron's output and its input. This non-linear function (also known as the activation function) can be a simple Relu function or a complex Izhikevich model in a Spiking Neural Network (SNN).

The bulk of the computation power in the inference stage is spent on processing neurons' inputs, calculating their outputs, and data movement between the layers. The complexity of this computation depends on the connections between the layers, bit resolutions of the weights, and activation functions. For example, 16-bit fixed-point wights result in less expensive MAC operations than 32-bit float weights, or crop-and-copy connections in UNets increase the memory utilization and require more complex scheduling.

Modern deep learning models enjoy architectures with large numbers of parameters, increasing numbers of layers, and irregular connections between non-adjacent layers. Although these architectures provide higher accuracies for their models, they also increase the model's implementation complexity on hardware, particularly for scale-out and power-limited systems. Moreover, current trends in DL indicate that future DNNs will have architectures closer to their biological counterparts with further complex neuron models and more connections between non-adjacent layers. Therefore, realizing DL accelerators on resource-limited processing platforms will become an even more challenging task.

A tolerable solution is to take advantage of the DL methods for retuning models and reshape the model for improving its hardware implementation on a target device. Currently, to accelerate a DL model's inference on a custom platform, DL designers pass an abstract model to hardware designers to create a functionally identical and lossless realization of the model on the hardware. Any modifications in this abstract will change the model's behavior and output. Therefore, any implementation optimization that entails reshaping the DNN architecture requires additional steps in the training stage to retune the model to ensure the quality of its output.

This thesis shows that a co-design approach can lead to creating better models with lower computational costs. In a co-design approach, an existing model is reshaped to optimize its

implementation on a target platform. Then, this modified model is retuned to ensure its output quality. Thus, the co-design process can include iterations of model reshaping and retuning until both output quality and hardware cost reach their desired targets.

The rest of this chapter presents the outline and motivations of the remaining chapters.

1.1 Reshaping Spiking Neural Networks

SNNs are the next generation of Artificial Neural Networks (ANNs) that utilize an event-based representation to perform more efficient computation. Most SNN implementations have a systolic array-based architecture assuming that high sparsity in spikes will significantly reduce computing in their designs. This chapter shows that this assumption does not hold for applications with signals of the large temporal dimension. We develop a streaming SNN (S2N2) architecture that can support fixed-per-layer axonal and synaptic delays for its network. Furthermore, we introduce a change in current SNN models to reshape their neuron's processing scheme for replacing MAC operation with ACC operations to optimize our accelerator's computational cost. We then show that current SNN models can preserve their accuracies after the retuning.

S2N2 is built upon FINN [9] and thus efficiently utilizes FPGA resources. By avoiding the tick-batching and replacing MAC operations with ACC operations, a stream of RF samples can efficiently be processed by this accelerator, improving the memory utilization by more than three orders of magnitude.

In addition, this chapter introduces novel SNN models for automatic modulation classification and shows how these radio frequency applications match our S2N2 computational model.

1.2 Filtering Noise in Neuromorphic Vision Data

Neuromorphic vision sensors are an emerging technology inspired by how the retina processing images. These sensors are a perfect match for SNNs to process their output.

A neuromorphic vision sensor only reports when a pixel value changes rather than continuously outputting the value every frame as is done in an ”ordinary” Active Pixel Sensor (ASP). This move from a continuously sampled system to an asynchronous event-driven one effectively allows for faster sampling rates; it also fundamentally changes the sensor interface. In particular, these sensors are highly sensitive to noise, as any additional event reduces the bandwidth and thus effectively lowers the sampling rate.

This chapter introduces a novel spatiotemporal filter with $O(N)$ memory complexity for reducing background activity noise in neuromorphic vision sensors. To design this filter, we study the noise characteristics in neuromorphic sensors and provide a novel filter with improved computational cost and more capabilities compared to previous works.

Our design consumes $10\times$ less memory and has $100\times$ reduction in error compared to previous designs. Our filter is also capable of recovering real events and can pass up to 180% more real events.

1.3 Reshaping Residual Neural Networks

ResNets employ skip connections as identity shortcut connections between the input and output of each residual block to ease the training of deeper networks. These connections are valuable for training to deal with the vanishing gradient problem and overcoming the saturation issue in very deep networks. However, skip connections also increase the implementation complexity in hardware. In particular, they are more problematic for inference accelerators on resource-limited platforms.

The hardware architecture has two options to implement skip connections: 1) keep the residual block data in the block's input buffer while the residual block is being processed, preventing it from receiving new input and degrading the pipeline performance, or 2) buffer this input elsewhere, forcing higher utilization of on-chip/off-chip memories, requiring larger memory bandwidth, and additional control logic.

This chapter introduces SKIPTRIM, a reshaping method for the ResNets. In this method, skip connections in smaller ResNets are entirely removed, then through a retuning technique, these reshaped networks can regain their accuracy.

For the larger ResNets with deeper architectures, we introduce a reshaping technique to reposition the skip connections for allowing a specific layer merge that can benefit the hardware implementation. We then show that our reshaped models can regain their accuracy after the retuning.

SKIPTRIM decreases the BRAM utilization by 20% by pruning the skip connections for smaller ResNets, and 16% by shortening them for larger ResNets on an FPGA with minimal to no loss in the model accuracy.

1.4 Auto Tuning the Learning Rate

Training a neural network is a complicated and time-consuming task that involves adjusting and testing different combinations of hyperparameters (the training parameters). After reshaping a DL model, hyperparameters may require readjustments which can create a bottleneck in the co-design process.

One essential hyperparameter for training DNNs is the learning rate, which balances the magnitude of changes at each training step. This hyperparameter has dependencies on both the model architecture and other training parameters such as the optimizer. For example, a reasonable learning rate value for a network can change by quantizing the network parameters.

This chapter introduces an Adaptive Scheduler for Learning Rate (ASLR) that significantly lowers the tuning effort since it only has a single hyperparameter. ASLR produces competitive results compared to the state-of-the-art for both hand-optimized learning rate schedulers and line search methods while requiring significantly less tuning effort. Furthermore, our algorithm's computational cost is trivial and can be used to train various network topologies included quantized networks.

Chapter 2

Reshaping Spiking Neural Networks

Artificial Neural Networks have shown remarkable success in large-scale image and video recognition [113, 117], speech recognition [11, 36], radio signal classification [95], and many other application domains [46, 63]. Spiking Neural Networks (SNNs) use an event-based model that better mimics biological neurons [33, 77] with the goal of providing high prediction accuracy while using minimal energy [118]. Recent advancements in SNN architecture design and training methods show promise in matching the accuracy of non-spiking ANNs [3, 8, 29, 115] and the potential to out-perform a similar-sized non-spiking ANN [17]. However, much work remains until we fully uncover the potentials of SNNs [118].

A conventional neuron model assumes every input requires calculation and performs N operations, e.g., multiplying and accumulating N input values with N weights (and an optional bias - see Equation 2.1). A typical convolutional layer in a modern feedforward neural network includes many neurons with an equal number of inputs (fan-in). This architecture creates patterns suitable for massively parallel implementations. Frameworks such as FINN [9], fpgaConvNet [132], and Eyeriss [12] provide efficient implementations of this architecture on FPGAs.

Conversely, event-based neural networks reduce wasted computation by only processing received events. For example when an event-based neuron with fan-in=N receives $M < N$ events, calculating the input only requires M operations (Equation 2.2). This assumes a certain amount of sparsity and requires dynamic handling of events. This sparsity creates a run-time dependency

based on the input data and induces unpredictable and potentially irregular memory accesses. Therefore exploiting parallelism in SNN is more challenging than CNNs, DNNs, and other more traditional neural networks.

Previous works such as IBM TrueNorth [3], Intel Loihi [19], SpiNNaker [96], and BlueHive [87] have shown that processing SNN events can be efficiently implemented in custom hardware for both training and inference. Neurogrid [8] uses a mixed analog-digital approach for simulating large-scale spiking models and Minitaur [90] and SpinalFlow [89] describe inference accelerators for SNNs. Event processing is either done by encoding and storing events in a buffer to be processed in a systolic fashion (tick-batching) [3, 19, 89, 90] or a spike-routing mechanism is used to prevent deadlocks [8, 87, 96].

In this work, we introduce a streaming accelerator for spiking neural networks, S2N2. Our design efficiently supports both axonal and synaptic delays for feedforward networks with interlayer connections. We show that because of the spikes' binary nature, a binary tensor can be used for addressing the input events of a layer. We describe the condition when addressing events with a binary tensor, and no tick-batching (streaming) can provide a better memory utilization compared to encoding events and tick-batching. We show that this condition depends on the input's sparsity (more detail in Section 2.2.3) and holds, for example, applications, in particular for Radio Frequency (RF) applications.

We use the FINN framework [130] as our baseline and extend it with new functions for supporting our event-based processing of SNNs. Our proposed changes can maintain the high throughput of FINN and provide an efficient streaming implementation for SNNs by benefiting from FINN's optimized utilization footprint.

We also propose novel example applications for SNNs in the RF domain that can benefit from S2N2's streaming architecture. By looking at RF samples as events in In-phase and Quadrature (I/Q) plane, RF samples can be turned into highly sparse events as input to a SNN. RF inputs available in RF datasets [94, 95] have a large temporal dimension, and a SNN designed for classifying these inputs can efficiently be implemented in S2N2.

In addition, our design is tested by using some of the published applications for SNNs in the image classification domain [53, 115]. In order to adopt these previously published spiking networks to S2N2, we propose new architectural changes in these networks and show that modified networks can maintain their accuracy after re-training with new hyperparameters.

Our contributions can be summarized as following:

- We introduce a new streaming architecture, S2N2, for accelerating SNNs on FPGA platforms.
- We describe how to reduce the memory utilization for inputs with a large temporal dimension.
- We propose novel applications for SNNs in the RF domain that can benefit from our streaming architecture.
- We release our code as open-sourced to enhance accessibility and aid in future comparisons of our work [1].

The remainder of the paper is organized as follows. In Section 2.1, we introduce SNNs in more depth and review the previous work on SNN FPGA implementations. S2N2 is described in detail in Section 2.2 and we demonstrate its advantages and implementation results for RF applications in Section 2.3. Additionally, Section 2.4 applies the S2N2 architecture to previously published SNNs for image classifications. We conclude our work in section 2.5.

2.1 Spiking Neural Network

Spiking neural networks are the third generation of ANNs developed to process information more similar to biological neural networks [77, 118]. In these networks, neurons propagate information by using spikes. The information is coded into the rate and time-of-arrival of the spikes.

[1] github.com/arkhodamoradi/s2n2

Figure 2.1 shows an example comparison between a frame-based input and an event-based input with rate-coded spikes. In general, input to each layer in a non-spiking neural network is a tensor of values (a multi-channel matrix). In contrast, in a SNN, inputs to each layer are events that have temporal and spatial positions. The temporal dimension of the input in SNNs consists of several "*ticks*". A tick is the minimum unit of time in a SNN that a neuron evaluates its input, updates its potential, and, depending on its model parameters, may generate a spike in its output.

For a more clear comparison, we look at the operations required for evaluating inputs in non-spiking and spiking neurons. Input to a non-spiking neuron is calculated as follows:

$$I = \sum_{i=0}^{N} w_i x_i \tag{2.1}$$

Here, x_i are the input values and w_i are their associated weights and bias is not shown.

While input to a spiking neuron at tick=t is calculated as following:

$$I_t = \sum_{i \in S_t} w_i \tag{2.2}$$

Here, S_t is the set of inputs to the neuron that have a spike at tick=t, and w_i are the weights associated with those inputs.

With sparsity in input spikes, Equation 2.2 requires fewer and simpler accumulation operations compared to the fixed number of MAC operations required in Equation 2.1. However, Equation 2.1 is more suitable for applying techniques such as loop-unrolling for exploiting parallelism. In addition, Equation 2.2 requires memory to store S to keep track of input spikes. Later in this work, we provide solutions to efficiently implement Equation 2.2 on custom hardware.

In a conventional ANN, an activation function of a neuron defines the output of that neuron given an input. In SNNs, neuron's output and evaluation of neuron potential are governed by *neuron's model*.

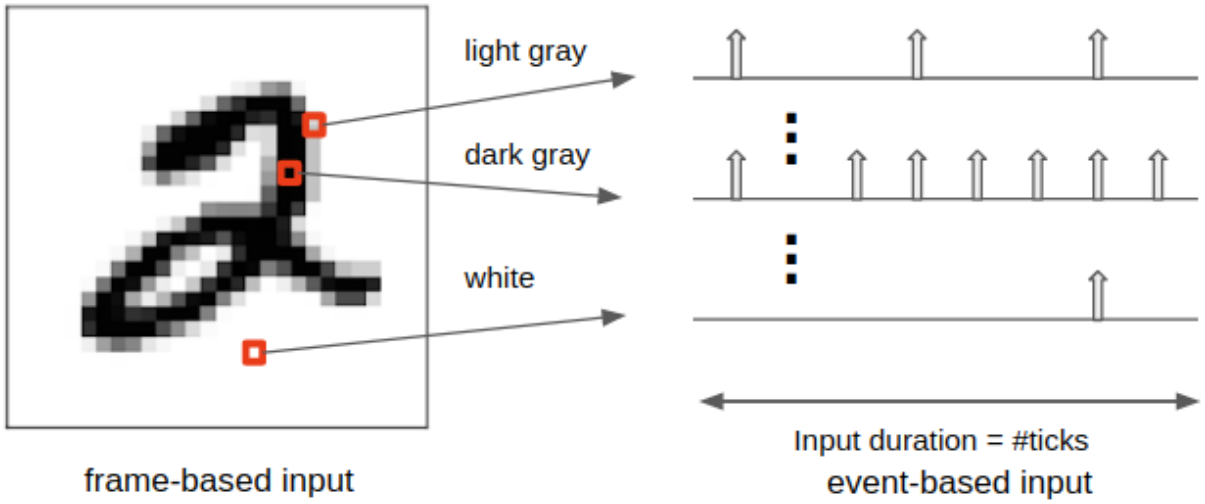


Figure 2.1. A frame-based input (on the left) is a matrix of numbers. An event-based input (on the right) includes trains of spikes. In this example, the number of trains is equal to the number of pixels in the frame. The duration of the spike trains is equal to the number of ticks. At each tick, up to one spike can exist in each train.

Neuron models used in SNNs are biologically plausible models that are computationally more powerful units compared to activation functions used in non-spiking networks [33]. These models are capable of extracting the temporal information embedded in their input and perform more complex tasks [77].

Although more complex mathematical models such as Izhikevich [50] and Hodgkin–Huxley [44] can accurately model a biological neuron's behavior, current training methods for SNNs are not geared to train these complex models [118]. For now, simpler models such as Integrate and Fire (IF) and Leaky Integrate and Fire (LIF) are more prevalent in current SNN applications. In this work, we use a LIF model with one internal parameter.

2.1.1 LIF Model

Leaky Integrate and Fire (LIF) model is a neuron model widely used in SNN applications [53, 64, 97, 115, 143]. LIF model memorizes its past inputs by adding every input to its membrane potential and uses a leak (decay) parameter to forget them. This leak parameter is reflecting the diffusion of ions that occurs through the membrane when some equilibrium is not reached in the

cell:

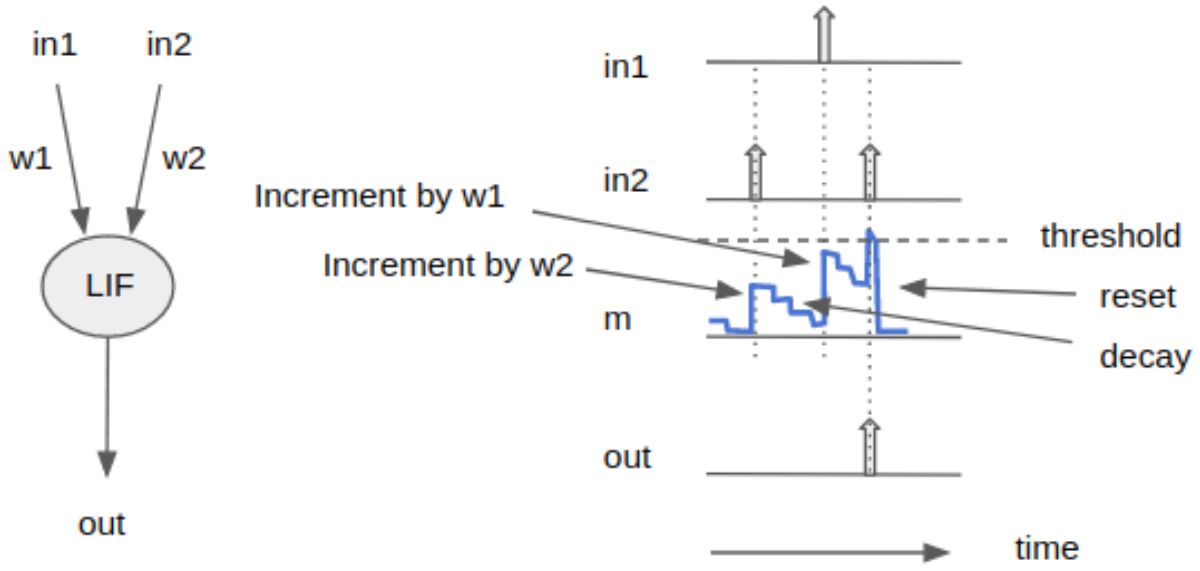


Figure 2.2. LIF neuron with two inputs. An incoming spike increases the membrane voltage by the weight associated with its connection. A decay parameter decreases the membrane potential, and if this voltage passes a threshold, it resets to a preset value, and the neuron generates a spike at its output.

$$m_t = (1 - \text{out}_{t-1}) * d * m_{t-1} + I_t \quad , \quad 0 < d < 1 \tag{2.3}$$

$$\text{out}_t = \begin{cases} 1, & \text{if} \quad m_t > T \\ 0, & \text{ow} \end{cases} \tag{2.4}$$

Here, $d \in (0,1)$ is the decaying leak parameter, T is the threshold, m_t is the membrane voltage at tick$= t$, and I_t is the input from Equation 2.2. The term $(1 - \text{out}_{t-1})$ in Equation 2.3 is the reset mechanism that sets the membrane voltage to zero if neuron fires a spike in its output. This mechanism is illustrated in Figure 2.2.

Generally, training LIF neurons is done by treating the threshold (T) and decay (d) as non-trainable hyperparameters.

2.1.2 Propagation Delays in Neuron

As shown in Figure 2.3, a biological neuron has different components. Simply, nerve impulses are received by dendrites and processed by the nucleus. Impulses generated by the neuron travel through the axon and are distributed through synapses.

This process has two propagation delays: 1) *Axonal delay* that is the time required for an action potential to travel from soma to synapses through the axon. 2) *Synaptic delay* that is the time interval required for a neurotransmitter to be released from a presynaptic membrane distribute across the synaptic cleft and received by the post-synaptic membrane.

Supporting these propagation delays in implementation can increase the complexity of the design. Hence, only a few previous works support these delays (more detail in the next section).

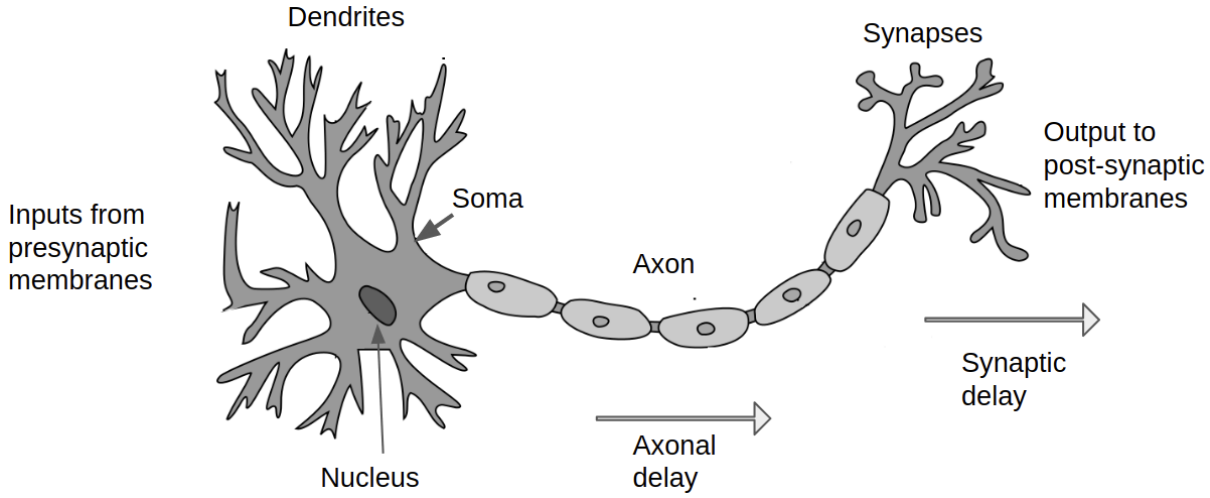


Figure 2.3. Illustration of a biological neuron. Dendrites receive inputs from presynaptic membranes to soma. The nucleus reacts to the received signals and may produce an action potential, which then has to go through the axon and distribute to post-synaptic membranes through Synapses.

2.1.3 Custom SNN Implementations

Analog [70,71,124], digital [3,13,19,87,89,90,96,126], and mixed-analog-digital [8,88] accelerators for SNNs have been described in the literature.

Analog realizations [70, 71, 124] are based on memristive technology [122] and have to deal with latency, density, and variability issues related to this technology [2]. In an other work [88], in addition to a memristive-based analog module, a digital module is used to route events and update receptive neurons. Neurogrid [8] does not use memristive technology in its analog module and increases parallelism by using a digital router for its events. In this work, we introduce a digital implementation for SNNs, and therefore we do not compare our work with analog realizations.

Large scale custom chip implementations such as Intel Loihi [19] with 4,096 on-chip cores and 1,024 neural units per core, SpiNNaker supercomputer [96] with 57,600 chips and 1,036,800 processors each capable of simulating 1,000 neurons, and IBM TrueNorth [3] with 4,096 cores and supporting one million neurons are designed with synaptic delay support. These implementations are designed to support a mesh of neurons with no particular topology. This is done by using advanced routers and schedulers. For example, Loihi uses six bits for the synaptic delay and two independent physical routing networks for core-to-core multicast. And events in SpiNNaker are coded to AER [78] packets (including timestamp, position, polarity, and debugging bits) and are source coded, meaning that the destination of each neuron has to be stored for routing the packets. TrueNorth has its own packet coding scheme, including the address of the core, axon index, tick number, and debugging flags. It buffers the events and uses a scheduler for processing events at specified ticks for supporting the synaptic delay.

Previous FPGA implementations of SNNs took a similar approach. BlueHive [87] is a 4-FPGA system and supports 64k Izhikevich [50] neurons per FPGA. BlueHive uses a routing system for events and 16 FIFOs for queuing events for 16 different synaptic delays with 1 millisecond granularity. Minitaur [90] encodes its events into five bytes, four bytes for timestamp and one byte for layer index. It supports a fixed axonal delay by buffering its events. In some other implementations routing and queuing is done without supporting synaptic or axonal delays [13, 32, 89, 126]. Because of queuing, parallelism in these works is done when an event is processed. Each event has a number of destinations, and upon processing an event, all of

Table 2.1. A comparison between S2N2 and previous works.

Architecture	Technology	Purpose	Supported Topology	Supported Propagation Delay	Required Complexity for supporting delay
Loihi [19]	custom chip	training and simulation	general mesh	synaptic	two separate physical routers
SpiNNaker [96]	custom chip	simulation	general mesh	synaptic	AER packets+router
TrueNorth [3]	custom chip	simulation	general mesh	synaptic	per-chip scheduler
BlueHive [87]	FPGA	simulation	general mesh	synaptic	16 FIFOs with 1ms granularity
Minitaur [90]	FPGA	accelerator	general mesh	fixed axonal	tick-batching and sorting
SpinalFlow [89]	FPGA	accelerator	feedforward	none	tick-batching (without supporting delays)
[126]	FPGA	accelerator	small and dense	none	N/A
[13]	FPGA	accelerator	feedforward	none	tick-batching (without supporting delays)
[32]	FPGA	accelerator	feedforward	none	tick-batching (without supporting delays)
[52]	FPGA	accelerator	feedforward	none	tick-batching (without supporting delays)
S2N2	FPGA	accelerator	feedforward	synaptic+axonal	streaming

its destinations (membrane potentials) are incremented by their associated weights in parallel. Routers and schedulers are used to prevent deadlocks and data hazards while processing events from different queues with the same destinations. A comparison is provided at Table 2.1.

In the next section, we argue that by considering the network topology, for a feedforward network with interlayer connections, fixed-per-layer axonal and synaptic delays can be supported without extra FIFOs, schedulers, and separate routing networks.

2.2 Streaming Spiking Neural Networks (S2N2)

To explain the streaming architecture of S2N2, we first look into the coding scheme used for storing events in input buffers. And explain the condition when a binary tensor can utilize less memory. We then explain how feedforward SNNs with interlayer connections can support fixed-per-layer synaptic and axonal delays without requiring schedulers and separate routing systems.

2.2.1 Input Buffer - Memory Utilization

As shown in Figure 2.1, a spiking input has a temporal duration with a total number of ticks (time units). In tick-batching, all the events for the entire duration of input are buffered and processed in a systolic implementation [89].

Let's look at the input events in a layer of a feedforward network. Assuming S being the total number of inputs to the layer, and T the total duration of the input, to encode events, we need $\log_2 S$ bits for addressing the position and $\log_2 T$ bits for addressing the tick number of each event. Assuming sparsity in the incoming events, the layer can receive up to $p * ST$ events when $p = 1 - sparsity\_ratio$ and $p \in (0, 1)$. Therefore we need a buffer of size:

$$\text{buffer size in bits} = pST \log_2 ST \tag{2.5}$$

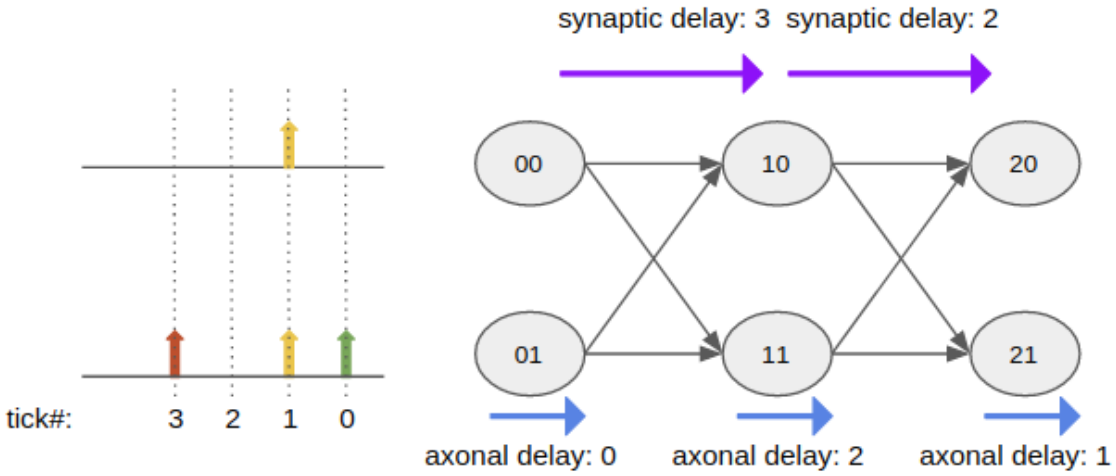
On the other hand, we can use a binary tensor to address the input events, ones for when there is an event, and zeros otherwise. In this case, we need ST-bits to store addresses in a binary tensor. Buffering encoded events requires less memory compared to a binary tensor if:

$$p \log_2 ST < 1 \tag{2.6}$$

This can be a tight condition on input's sparsity. E.g., for a layer with an input tensor of size $64 \times 16 \times 16$ with a total duration of 16 ticks, only for $p < \frac{1}{18} = 5.5\%$ or 94.5% sparsity for input, buffering encoded events uses less memory compared to a binary tensor of size 2^{18} bits. In this example, as soon as the input's sparsity drops below 94.5%, Equation 2.6 is not satisfied, and the binary tensor requires less memory. In Sections 2.3 and 2.4, we show that Equation 2.6 can not be satisfied for our applications.

2.2.2 Fixed-Per-Layer Propagation Delays

As mentioned before, synaptic delays are realized in a limited number of previous work. Custom chips [3, 19, 96] queue their events and use complex routing and scheduling systems to process events at the correct tick with an appropriate delay. In FPGA implementations, multiple FIFOs are used to support synaptic delays with large granularity (1 millisecond) [87]. These implementations support different topologies of spiking networks. And [90] supports feedforward networks with fixed axonal delays by buffering and sorting its encoded events.



tick #	in(00)	in(01)	out(00)	out(01)	in(10)	in(11)	out(10)	out(11)	in(20)	in(21)	out(21)	out(22)
0	0	1	0	1								
1	1	1	1	1								
2	0	0	0	0								
3	0	1	0	1	1	1						
4					2	2						
5					0	0	1	1				
6					1	1	1	1				
7							0	0	2	2		
8							1	1	2	2	1	1
9									0	0	1	1
10									2	2	0	0
11											1	1

Figure 2.4. Top: A simple 3-layer network with fixed-per-layer axonal and synaptic delays. Inputs to each layer are binary vectors, and "1"s are for spikes. Input to a neuron is the sum of all inputs, and all weights are equal to one. Bottom: the flow of the input through the network. E.g., network input at tick=1 (color-coded) is received by both neurons in the first layer. With no axonal delay, they each produce one spike at their outputs at the same tick. The second layer receives this input (same color code) with a synaptic delay (3 ticks). At tick=4, both inputs to each neuron in the second layer have spikes. Hence their inputs are equal to 2.

Feedforward SNNs with interlayer connections have a specific topology that can be exploited for supporting fixed-per-layer synaptic and axonal delays with a reduced implementation cost. As shown in [133], temporal coding is still possible with fixed propagation delays. Figure 2.4 shows a simple 3-layer network with fixed-per-layer axonal and synaptic delays, meaning that all neurons in one layer have the same axonal delay and the same synaptic delay. For the sake of simplicity, neurons in this network spike if they receive an input larger than zero, and all

weights are equal to one. Input to each neuron is the sum of all inputs.

The bottom part of Figure 2.4 shows how spikes spread through the network under axonal and synaptic delay conditions. Input to each layer is a binary vector, and spikes are represented by "1"s. Weights are equal to one, and input to a neuron is the sum of weights for connections with a spike. E.g. at tick=4, both inputs to neuron 10 have spikes and $\text{in}(10) = 2$. Previous works with propagation delay support [3, 19, 87, 90, 96] support this with different complexities (see Table 2.1).

tick #	in(00)	in(01)	out(00)	out(01)	in(10)	in(11)	out(10)	out(11)	in(20)	in(21)	out(21)	out(22)
0	0	1	0	1	1	1	1	1	2	2	1	1
1	1	1	1	1	2	2	1	1	2	2	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	1	0	1	1	1	1	1	2	2	1	1
4												
5												
6												
7												
8												
9												
10												
11												

Figure 2.5. With fixed-per-layer propagation delays in the example network shown in Figure 2.4, we can process inputs and outputs of all layers assuming no delay and push all the delays to the end. Then an accumulated delay (shown in purple and blue) can be added to the network output.

However, because of the network topology, we can process all the layers, assuming no propagation delay, and push all the delays to the end. Then a total delay equal to all accumulated delays can be applied to the network's output as shown in Figure 2.5.

This practice can be applied to any structured feedforward network with only interlayer connections. In this case, we can support both synaptic and axonal delays without schedulers, extra FIFOs, and sorting mechanisms used in previous works.

2.2.3 Architecture

The streaming architecture of S2N2 is designed based on the FINN framework [130]. In the following, we first describe FINN's approach to implementing non-spiking and conventional

neural networks. We then describe our design to support the LIF model in FINN.

FINN framework: The original FINN paper [130] introduced a framework for building fast and flexible FPGA accelerators using a flexible heterogeneous streaming architecture. Exploiting a set of optimizations, FINN enables efficient mapping of binarized neural networks to hardware and supports fully connected, convolutional, and pooling layers. The second version of FINN described in [9], provides support for non-binary networks.

In the FINN architecture, a Sliding Window Unit (SWU) prepares the input by applying interleaving and implementing the image-to-column (im2col) algorithm. The output stream of a SWU feeds a Matrix Vector Threshold Unit (MVTU), which is the computational core for FINN's accelerator designs. This core is used in the implementations of both fully connected and convolution layers.

As shown in Figure 2.6, a MVTU has several Processing Elements (PE) that can generate output channels in parallel. Each PE has a number of SIMD lanes. If P_{FINN} be the number of PEs and S_{FINN} be the number of SIMD lanes, A P_{FINN}-high, S_{FINN}-wide tile matrix is processed at a time, inputs are mapped to different SIMD lanes and outputs are calculated in parallel by PEs. To accommodate this process, weights are also loaded from memory in tiles, and each PE takes a sub-tile of the weights to process its output.

All PE units have access to the input buffer inside the MVTU. The width of this buffer in bits is equal to the number of SIMD lanes multiplied by the activation bit width. For simplicity, only one row of this buffer is shown in Figure 2.6. The total number of rows in this buffer is equal to the ratio of *(kernel width* $\times$ *kernel height* $\times$ *#input channels)/#SIMD lanes*. Which makes the input buffer size equal to *(kernel width* $\times$ *kernel height* $\times$ *#input channels)* for 1-bit activation.

The process flow of one PE is shown in (Figure 2.7.upper). The accumulator is initialized to a preset value (usually zero), then input and PE's sub-tile of weights are loaded into SIMD lanes. SIMD lanes execute Equation 2.1 and accumulate the results in the accumulator. After processing all the inputs for the current output, the accumulator value is passed through the

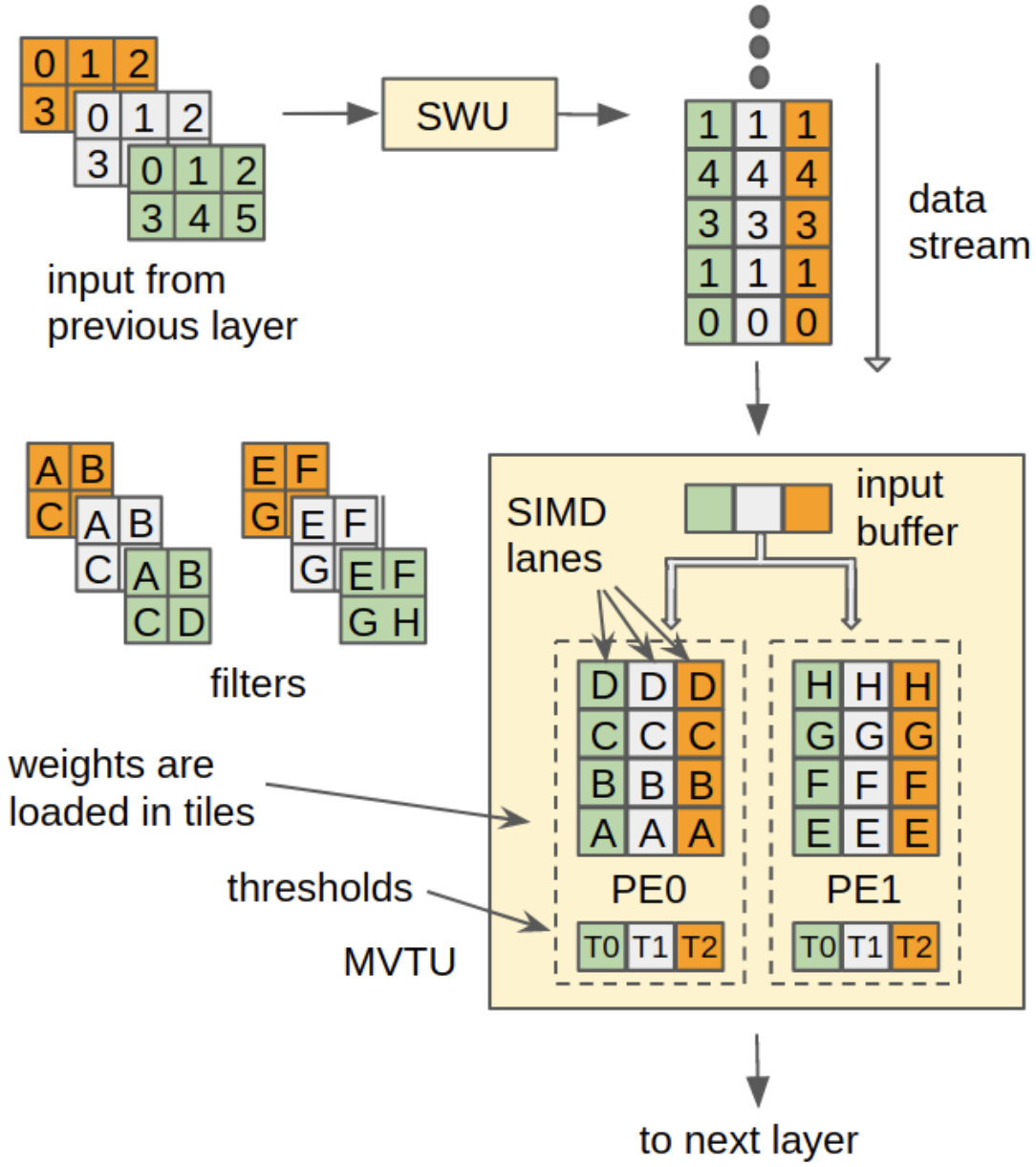


Figure 2.6. FINN [130] architecture. SWU interleaves the input by applying the image-to-column algorithm and feeds MVTU. Each PE inside MVTU processes one output channel and has a number of SIMD lanes that read from input channels and multiply the input by kernel weights in parallel.

activation for generating the output.

Implementing the LIF Model: S2N2's contribution to the FINN platform is by providing support for the LIF model in FINN's computational core, MVTU. In the following, we describe our design in detail and explain how to utilize FINN's architecture for initializing

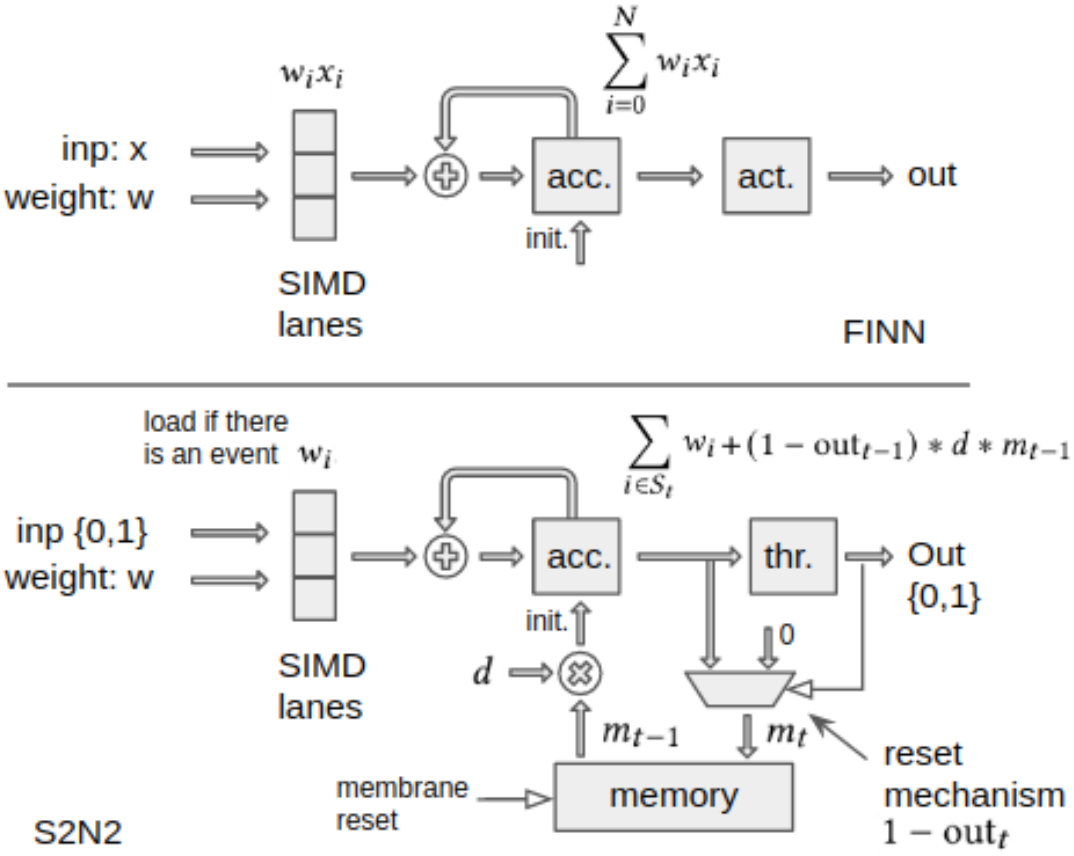


Figure 2.7. Upper: PE implementation in FINN [130]. Lower: PE architecture to support SNN in FINN.

membrane potentials for each input. In the next section, we describe a possible optimization for decaying membrane potentials without a multiplication operation to maintain FINN's high throughput.

As mentioned in Section 2.2.1, if the condition in Equation 2.6 is not met, using a binary tensor for addressing events has a lower memory utilization than encoding events. In Sections 2.3 and 2.4, we show that this condition does not hold for example applications. Therefore, we use the input as is (a binary tensor) for addressing events. In addition, because FINN is not a systolic implementation, and the input is processed in a streaming architecture, the size of the input buffer used in the MVTU can be smaller than the input [130]. E.g., for an input of size $I_{\text{W}} \times I_{\text{H}} \times I_{\text{Ch}}$ with a kernel size of $K_{\text{W}} \times K_{\text{H}}$, and 1-bit activation, the buffer size is equal to

$K_W \times K_H \times I_{Ch}$.

To support SNNs with the LIF model (Figure 2.7.lower), we initialize the accumulator by the previous membrane voltage stored in the on-chip memory, multiplied by the decay value, d. The SIMD lanes are programmed to use the input as the address of events and only load weights if there is a spike in that input position. This is exactly executing Equation 2.2. After executing all the operations required by Equation 2.2, the value stored in the accumulator is equal to Equation 2.3. This value can then be passed to a comparator to execute Equation 2.4. The result of this comparator is our output spike and is also used as the input to a selector for implementing the reset mechanism $(1 - \text{out}_t)$ and storing the correct membrane voltage back to the memory.

The MVTU in a FINN implementation has a control signal defining the number of runs per input. We use this signal to indicate the last tick for an input. This signal can be used to reset membrane potentials to zero if required.

In FINN, the MVTU is used for implementing both convolutional and fully connected layers. Similarly, with our proposed additions and modifications to the MVTU, both convolutional and fully connected layers for SNNs can be implemented with the MVTU shown in Figure 2.7.lower.

The pooling unit (PU) described in FINN [130] is a binary max-pooling layer. We chose to use binary tensors for addressing our spikes. Therefore we use the PU as is.

As illustrated in Figure 2.8, an event-based input has a temporal dimension that is divided into a number of ticks. To produce the classification output, the last layer in a SNN has one counter per label. Each counter keeps track of all the spikes received for that label. At the last tick for an input, the value of these counters can be fed to a function for determining the classification result (e.g., SoftMax). These counters are reset to zero at the last tick of each input.

As we explained earlier in this section, fixed-per-layer propagation delays in feedforward SNNs with only interlayer connections can be added as an accumulated delay to the output. Therefore, our proposed design can be used to implement such networks with fixed-per-layer

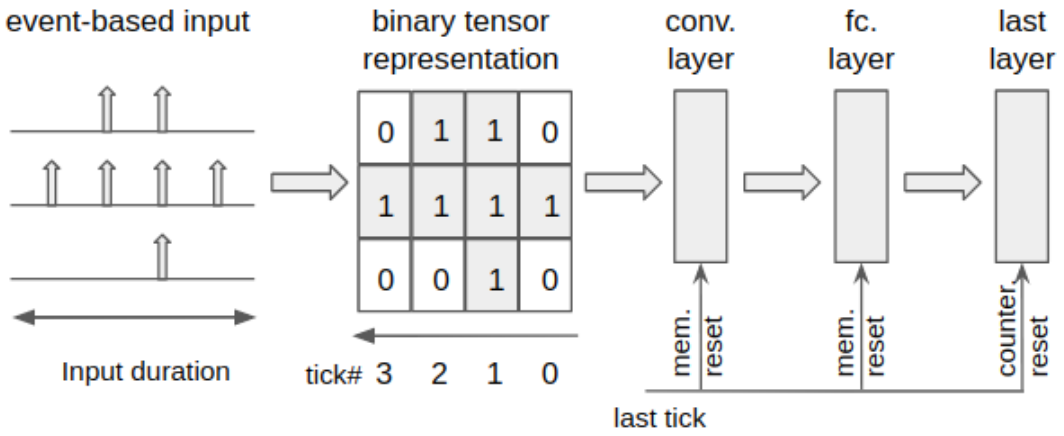


Figure 2.8. Binary tensor for addressing spikes in an event-based input. The last tick is used as a reset signal to reset membrane potentials (mem. reset) and counters at the last layer (counter reset).

axonal and synaptic delays. This design has no scheduler, and we do not queue encoded events. This gives us a number of advantages. 1) We can expand the parallelism of our design by processing events in parallel vs. sequential process in previous work. 2) By using a binary representation for addressing instead of addressing events by their position and tick number, we do not require a separate router. 3) we can support fixed-per-layer axonal and synaptic delays without a scheduler. 4) Addressing events with a binary tensor reduces our memory utilization when the condition in Equation 2.6 is not met.

2.3 S2N2 for Automatic Modulation Classification

Deep learning for Radio Frequency (RF) applications is a relatively new field. In particular, using SNNs for RF applications is barely touched in the literature. One of the RF applications suitable for ANNs is Automatic Modulation Classification (AMC). This important method can be used in radio fault detection, opportunistic mesh networking, dynamic spectrum access, and numerous regulatory and defense applications. Previous works have shown that ANNs can effectively perform modulation classification with high accuracy [73, 83, 94, 95].

This section introduces two new network architectures for AMC that are based on S2N2.

The novelty of these architectures is that the input is fed to the network as a stream of events in the In-phase/Quadrature (I/Q) plane. To our knowledge, these are the only neural networks that consume a stream of RF samples as an event-based input. In the following, we describe the datasets used for training and explain our networks' architecture.

Datasets: We use two RF datasets to train our networks. *RadioML.2016* [94] is a collection of 11 different modulations (8PSK, AM-DSB, AM-SSB, BPSK, CPFSK, GFSK, PAM4, QAM16, QAM64, QPSK, and WBFM). Each class has samples recorded at 20 different Signal to Noise Ratio (SNR) levels (from -20dB to 18dB in increments of 2dB). Each pair $\{$modulation, SNR$\}$ has 728 training examples, and Each training example is a time-series of 128 In-phase Quadrature (I/Q) sample pairs.

RadioML.2018 [95] is a collection of 24 different modulations (OOK, 4ASK, 8ASK, BPSK, QPSK, 8PSK, 16PSK, 32PSK, 16APSK, 32APSK, 64APSK, 128APSK, 16QAM, 32QAM, 64QAM, 128QAM, 256QAM, AM-SSBWC, AM-SSB-SC, AM-DSB-WC, AM-DSB-SC, FM, GMSK, and OQPSK). Each modulation class has samples recorded at 26 different SNR levels (from -20dB to 30dB in increments of 2dB). Each pair $\{$modulation, SNR$\}$ has 4096 training examples and Each training example is a time-series of 1024 I/Q sample pairs. Both datasets are publicly available [2].

Two time-series examples from RadioML.2018 are shown in Figure 2.9.left. These examples are 1024 I/Q sample pairs. In all of the previous work, inputs are tensors with same shape as these examples. E.g., for RadioML.2016, inputs are 2×128 float tensors, and in RadioML.2018, inputs are 2×1024 float tensors.

S2N2 is not a systolic implementation. Meaning, we can feed the network with a stream of events. Therefore, in our networks, we use the constellation of signals (shown in Figure 2.9.middle), and at each tick, we feed the network with one sample (Figure 2.9.right). Therefore our input is a stream of binary tensors.

To our knowledge feeding RF samples as events to a neural network has never been done

[2] https://www.deepsig.ai/datasets

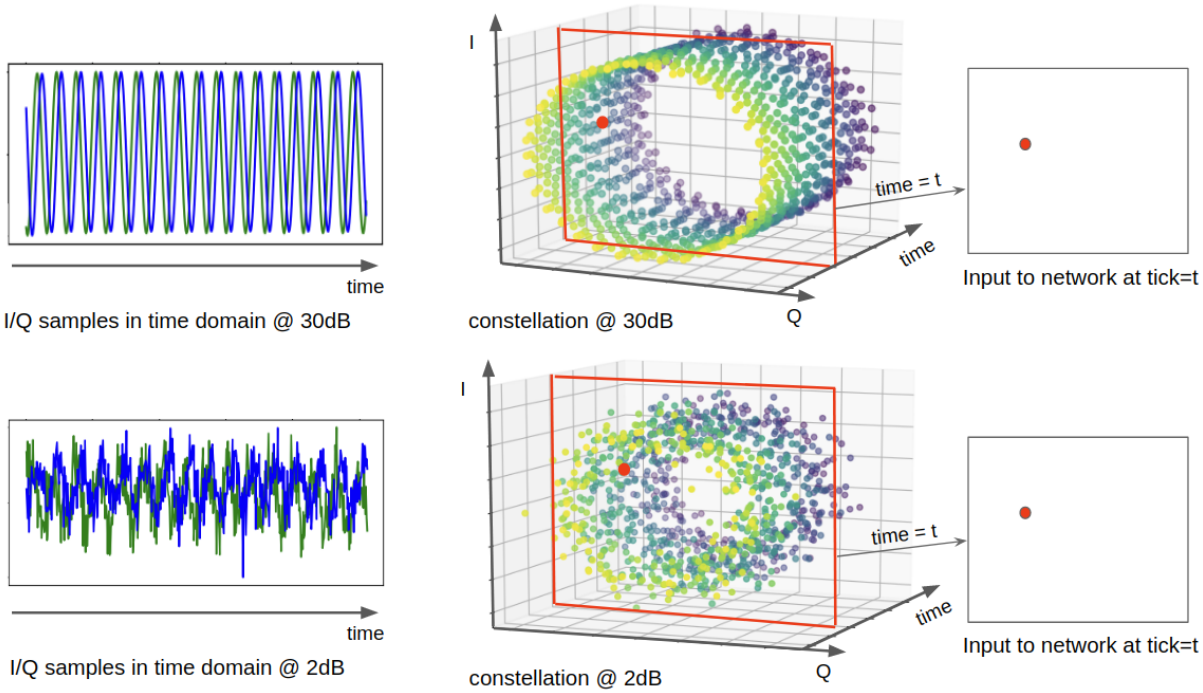


Figure 2.9. Examples of AM-DSB class from RadioML dataset [95]. On the left, two examples of AM-DSB I/Q samples are shown at 30dB and 2dB SNR at top and bottom, respectively. The middle illustrates the constellations of the same examples. On the right, input to the network at time (tick)=t is shown. Input to our networks are samples as events in I/Q plane.

before. The only work on using SNNs for AMC is a preliminary investigation done by NASA [57] that implements a two-layer SNN in MATLAB for classifying three noise-free modulations (BPSK, QPSK, and 8PSK). In NASA's work, inputs are 8-bit images of constellations.

Feeding a neural network with RF samples as events come with two benefits. 1) Although we and all the previous works use recorded data, in a real-world setup, our network can consume RF samples one-by-one in a stream. Other works have to buffer samples (e.g., 128 or 1024 samples) before taking them as input. 2) We can aggressively quantize the I/Q plane; therefore the input size (in bits) can get smaller. The following explains the I/Q plane quantization.

Examples in RadioML.2016 and RadioML.2018 are 128 and 1024 pairs of float numbers, respectively. We construct the I/Q plane by quantizing the pair using a uniform quantizer. This will reduce our input size. Figure 2.10 illustrates three examples from RadioML.2018: OOK, 64QAM, and 32PSK classes at 30db, 16dB, and 2dB SNR, respectively. To the right of these

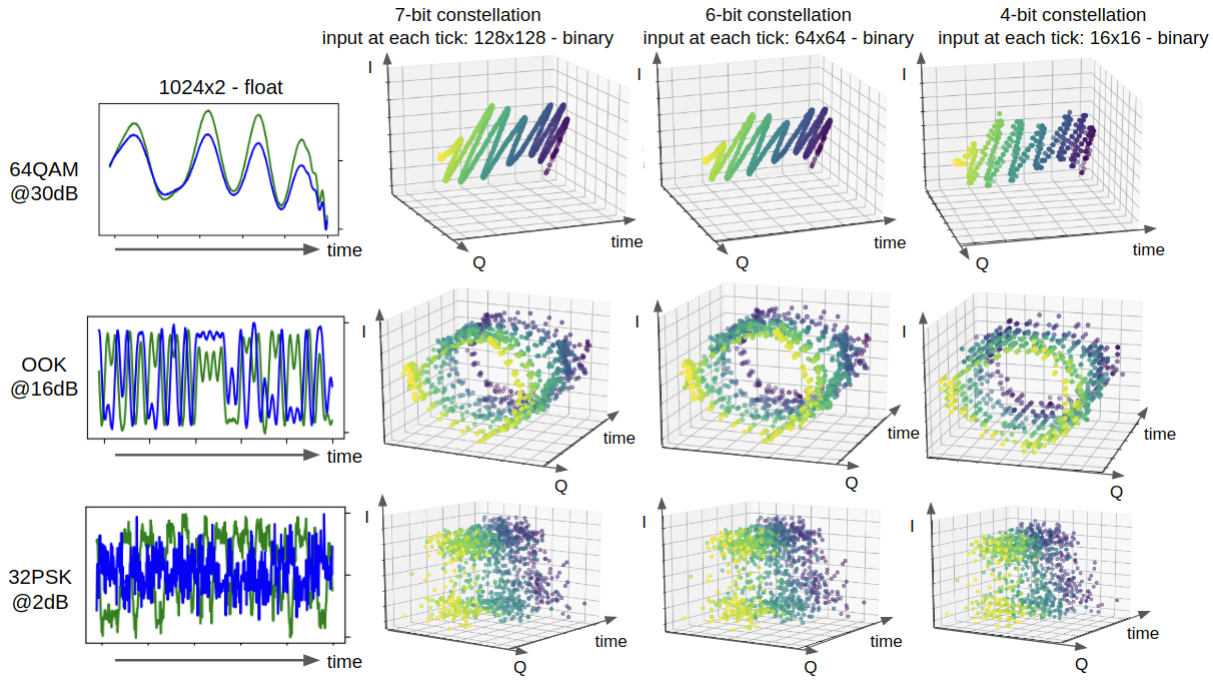


Figure 2.10. Applying quantization to the I/Q plane. Original examples from RadioML.2018 [95] dataset are 1024 pairs of float numbers (left). In-phase and Quadrature values can be quantized for a smaller input tensor (first three columns from the right). At each tick, we feed one slice of the quantized constellation tensor to the network. The figure shows that the constellation shape is recognizable while I/Q plane is aggressively quantized.

examples, their constellations with quantized in-phase and quadrature values are shown. As it is shown, the shapes of the constellations are recognizable even at the lowest bit resolution. We used the 4-bit quantized constellations to train our networks.

Network Architecture for RadioML.2016

This network is a four layer architecture similar to the network described in [94] with different number of kernels and LIF model for activation (Figure 2.11).

Inputs to each layer are binary tensors. We used 90% of the dataset for training and 10% for validation. For training, we used the method described in [108] as our baseline and changed the loss function to *smooth L1 loss*, and adjusted the hyperparameters. Throughout this chapter, we refer to this network as *S2N2_rf1*.

We achieved 91.7% Top-1 and 100% Top-5 validation accuracy using all SNR levels in

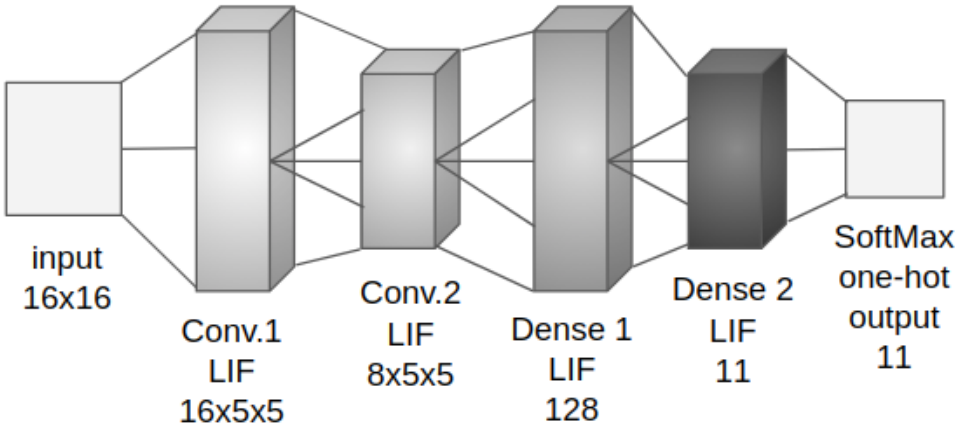


Figure 2.11. S2N2_rf1 architecture.

Table 2.2. Comparing validation accuracy and network size for S2N2_rf1.

Network	Input	Conv.1	Conv.2	Dense 1	Dense 2	Accuracy
[94]	128x2 32-bit	64x1x3	16x2x3	128	11	87.4%
S2N2_rf1	16x16 binary	16x5x5	8x5x5	128	11	91.7%

our training. A comparison between S2N2_rf1's size and accuracy with the previous work on RadioML.2016 is provided in Table 2.2.

Figure 2.12 illustrates the spike ratio in the input of each layer for S2N2_rf1. The first convolution layer (Conv.1) receives one event at each tick; this means that the spike ratio for this layer with an input of size 16×16 is $\frac{1}{16 \times 16} = 0.0039$.

As mentioned in Sections 2.1.3 and 2.2.1, previous works have used tick-batching and buffered encoded events. This means that for a total number of ticks=128, and input size of 16×16 at spike ration of 0.39%, according to Equation 2.5, tick-batching requires 1,917 bits to queue the input events. Because S2N2 is based on the streaming architecture of FINN [130], and only a portion of the input is buffered for processing. The size of this buffer used in MVTU is

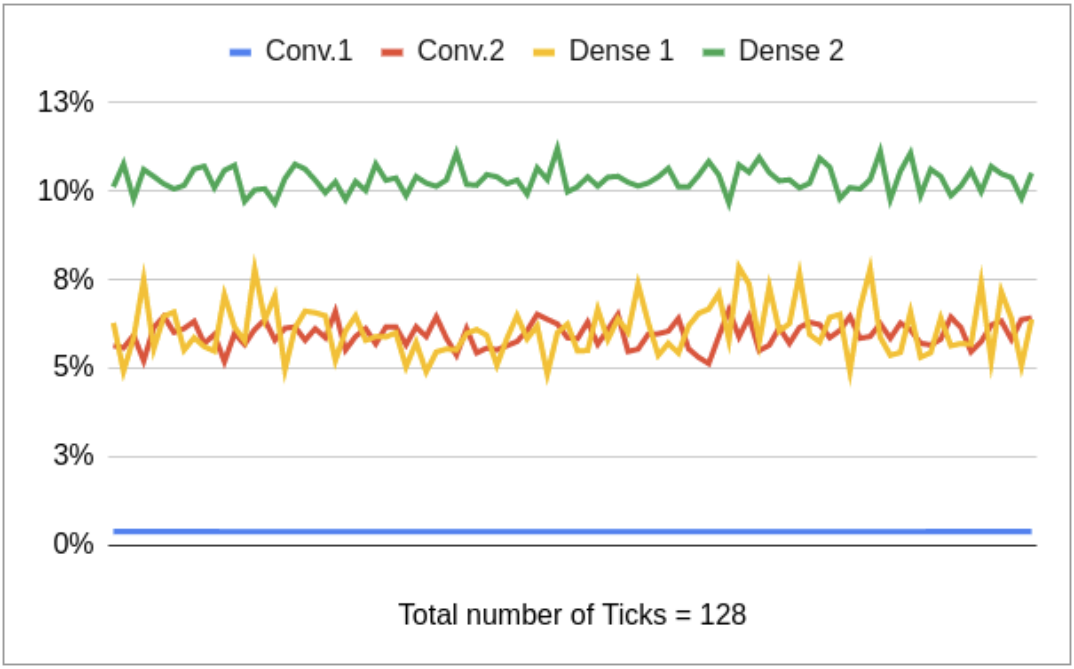


Figure 2.12. The ratio of spiking neurons in input to each layer of S2N2_rf1. Ratios are collected during classifying one input (128 ticks) with trained weights.

equal to *kernel size* $\times$ *#input channels*=25 bits for Conv.1 layer. In Table 2.3, we provide the same comparison for all the layers of this network. These results show that, on average, memory utilization for input buffers in S2N2_rf1 is improved by over three orders of magnitude.

Network Architecture for RadioML.2018

As mentioned in our introduction, training methods for spiking neural networks are not as mature as other ANNs. In particular, current training methods are evaluated on smaller networks, and simple datasets [32] and perform poorly when used for training very deep architectures [53, 115] and evaluated on more complex datasets [112]. Therefore we could not train a deep spiking network similar to the non-spiking networks used in previous works (VGG10 and Resnet33) [95, 129]. Instead, we chose a smaller network with only eight layers. We refer to this network as *S2N2_rf2*.

S2N2_rf2 architecture is shown in Figure 2.13. We used the same training script like the one we used for training S2N2_rf1 as the baseline. We then adjusted the script for the dataset

Table 2.3. Required memory for buffering input at each layer of S2N2_rf1 is compared with tick-batching (Equation 2.5).

Layer	#Ticks	Input Size	Maximum Spike Ratio	Buffer Size Tick-Batching	Buffer Size S2N2_rf1	Improvement
Conv.1	128	16×16	0.39%	1,917 bits	25 bits	×77
Conv.2	128	16×16×16	7%	697,304 bits	400 bits	×1,744
Dense 1	128	8×12×12	8%	212,337 bits	128 bits	×1,658
Dense 2	128	128	12%	27,526 bits	11 bits	×2,502

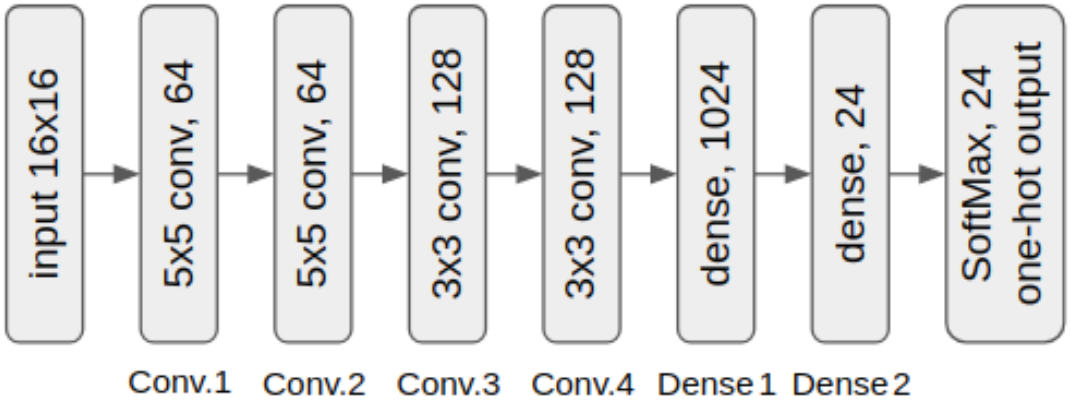


Figure 2.13. S2N2_rf2 architecture.

and its increased number of labels.

This network can achieve 68.5% Top-1 and 95% Top-5 validation accuracy on 24 classes in RadioML.2018 dataset. Table 2.4 compares our accuracy with two related non-spiking networks.

Although that S2N2_rf2 does not have a high accuracy compared to deeper and non-spiking networks, it is included in this work to provide a comparison between S2N2 architecture and tick-batching with regards to memory utilization. In particular, when larger RF inputs are used.

Figure 2.14 shows spike rations at the input of each layer of S2N2_rf2. These ratios are similar to the ratios in S2N2_rf1 (Figure 2.12). We expect that with future improvements in

Table 2.4. Comparing validation accuracy and network size for S2N2_rf2.

Network	Input	#Layers	Accuracy
ResNet [95]	1024x2 (32-bit)	33	95.5%
VGG [95]	1024x2 (32-bit)	10	88.0%
S2N2_rf2	16x16 (binary)	6	68.5%

training methods for deeper SNNs, similar spike ratios with no significant reductions will hold for a spiking network with a higher accuracy.

We use these ratios to show the efficiency of S2N2 for reducing the input buffer size at each layer. Even if our assumption does not hold, and in the future networks with lower spike ratios provide a higher accuracy, S2N2 is still more efficient at the minimum possible spike ratio; only one spike at layer's input (first row in Tables 2.3 and 2.5).

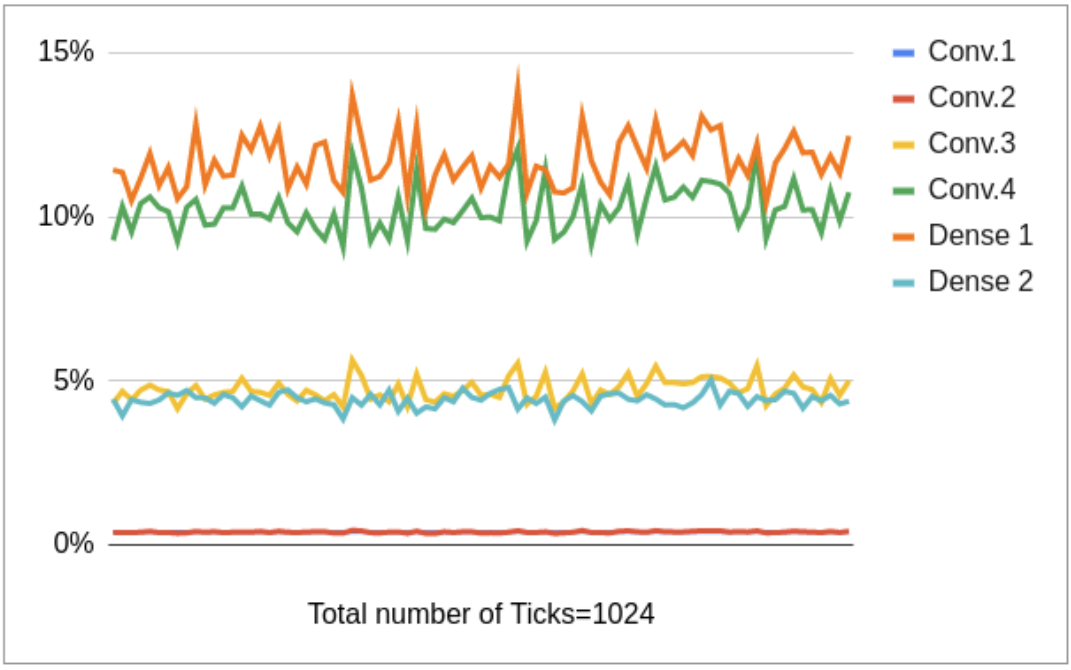


Figure 2.14. The ratio of spiking neurons in input to each layer of S2N2_rf2. Ratios are collected during classifying one input (1024 ticks) with trained weights.

Table 2.5 illustrates a comparison between input buffer sizes required for S2N2 and tick-batching. Equation 2.5 is used to calculate the buffer size for tick-batching. It is clear that for inputs with large temporal dimension, using a streaming architecture significantly reduces the memory utilization.

Table 2.5. Required memory for buffering input at each layer of S2N2_rf2 is compared with tick-batching (Equation 2.5).

Layer	#Ticks	Input Size	Maximum Spike Ratio	Buffer Size Tick-Batching	Buffer Size S2N2_rf2	Improvement
Conv.1	1024	16×16	0.39%	18,403 bits	25 bits	×737
Conv.2	1024	16×16×64	0.5%	2,013,266 bits	1,600 bits	×1,258
Conv.3	1024	12×12×64	6%	13,589,545 bits	576 bits	×23,592
Conv.4	1024	10×10×128	12%	37,748,736 bits	1,152 bits	×32,768
Dense 1	1024	10×10×128	14%	44,040,192 bits	1,024 bits	×43,008
Dense 2	1024	1024	5%	1,048,576 bits	24 bits	×43,690

Synthesis Results

We used Vivado-HLS™tool for evaluating S2N2_rf1 and S2N2_rf2 network architectures. To increase the throughput and reduce our DSP utilization, we used fixed-points for our parameters and trained both networks with a decay factor equal to 0.875 (d in Equation 2.3). This way, $d \times m_{t-1}$ in Figure 2.7 can be replaced by $(m_{t-1} - m_{t-1} >> 3)$.

We could fit S2N2_rf1 (smaller network) on a ZYNQ chip similar to the one used in the PYNQ development board. Because of the large size of S2N2_rf2, we selected the ZCU111 development board in our synthesis. This board is also used for implementing a non-spiking network for the same dataset [129].

Our results are shown in Table 2.6. The high BRAM utilization is due to the required memory for storing membrane potentials. Tick resolution indicates how fast RF samples can be consumed by the network. E.g., at each second, S2N2_rf1 can classify 173.6k examples from

Table 2.6. Synthesis results for S2N2_rf1 and S2N2_rf2 network architectures.

Network	Board	BRAM_18K	DSP48E	FF	LUT	Tick Resolution
S2N2_rf1	PYNQ	29%	5%	11%	52%	45 ns
S2N2_rf2	ZCU111	98%	¡1%	4%	24%	30 ns

the RadioML.2016 dataset (each example requires 128 ticks). And S2N2_rf2 can process 32.5k examples from the RadioML.2018 dataset (each example requires 1024 ticks).

2.4 Image Classification on S2N2

In this section, we provide an example network for image classification on MNIST dataset [3]. We used the method provided by DECOLLE [53] to convert MNIST dataset to trains of spikes [4]. We used a four-layer convolutional network similar to the one described in DECOLLE as our baseline.

Figure 2.15.left shows the network structure we used for image classification. We refer to this network as S2N2_cv. We applied two modifications to the original structure. First, as shown in the figure, in the original network, convolutional filters are applied to the membrane voltage. This means MAC operations similar to Equation 2.1. We modified the layer, and instead, we apply the convolutional filters on the input to have the sparse accumulations similar to Equation 2.2. Second, the neuron model used in the original work is a LIF model with two internal variables. We changed the model to the one variable LIF model described in Section 2.1.1.

S2N2_cv is trained with the original training script [4], and adjusted hyperparameters. It can achieve competitive results compared to other works (see Table 2.7).

Figure 2.16 shows the spike ratios at the input of each layer in S2N2_cv. The method for converting MNIST data to trains of spikes used in [53] converts each image to a $28 \times 28 \times 500$

[3] http://yann.lecun.com/exdb/mnist/

[4] https://github.com/nmi-lab/dcll

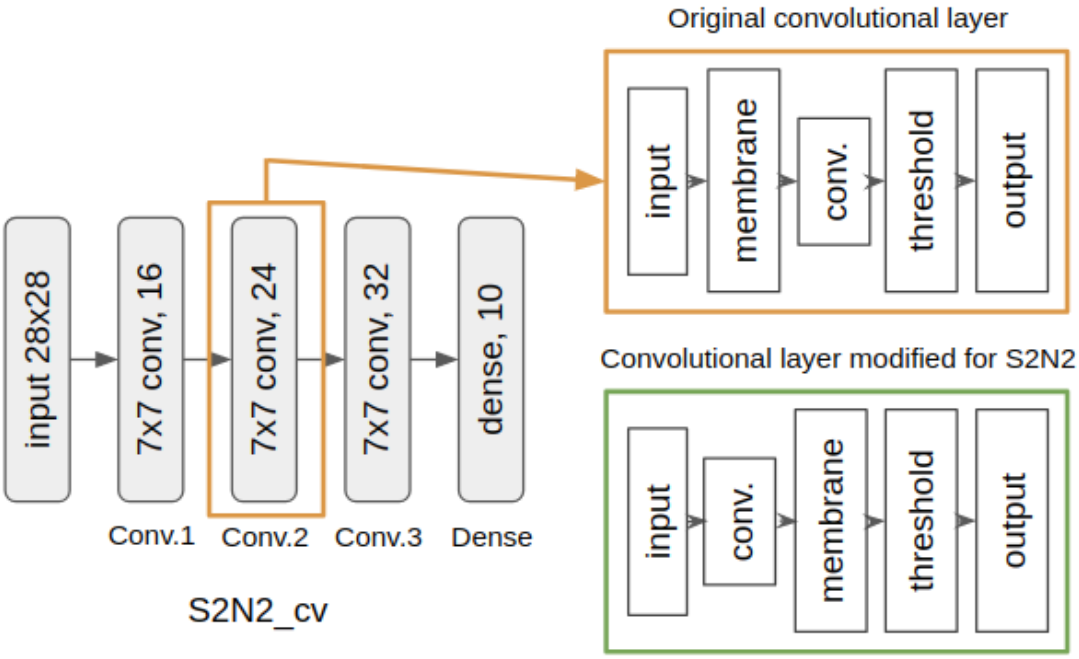


Figure 2.15. S2N2_cv structure. On the left, four-layer structure of the network. Orange box, original organization of one convolutional layer. Green box, convolutional layer modified for S2N2.

binary tensor of spikes. Unlike RF samples, input to the first layer can have more than one spike at each tick; therefore, for vision applications, the input is less sparse. Consequently, the ratio of spikes at each layer is higher than the layers in S2N2_rf1 and S2N2_rf2.

With higher spike ratios, the buffer size for storing encoded events in tick-batching rapidly grows. While the buffer size used in S2N2 is independent of the input's spike ratio. Table 2.8 shows a comparison between these two buffer sizes for S2N2_cv network.

Synthesis Results

S2N2_cv is evaluated with Vivado-HLS™tool. This network is relatively small, and we can fit it on the PYNQ development board. Our results are shown in Table 2.9.

To reduce our DSP utilization, we took a similar approach as what we did for training our two other networks and trained S2N2_cv with a decay factor equal to 0.875 (d in Equation 2.3). This way, $d \times m_{t-1}$ in Figure 2.7 is replaced with a shift and one subtractions $(m_{t-1} - m_{t-1} >> 3)$.

Table 2.7. Accuracy result of S2N2_cv on MNIST compared to similar SNNs.

Network	Architecture	Validation Accuracy
S2N2_cv	28x28-16c7-24c7-32c7-10	98.5%
[53]	28x28-16c7-24c7-32c7-10	98.0%
[115]	28x28-12c5-2a-64c5-2a-10c	99.3%

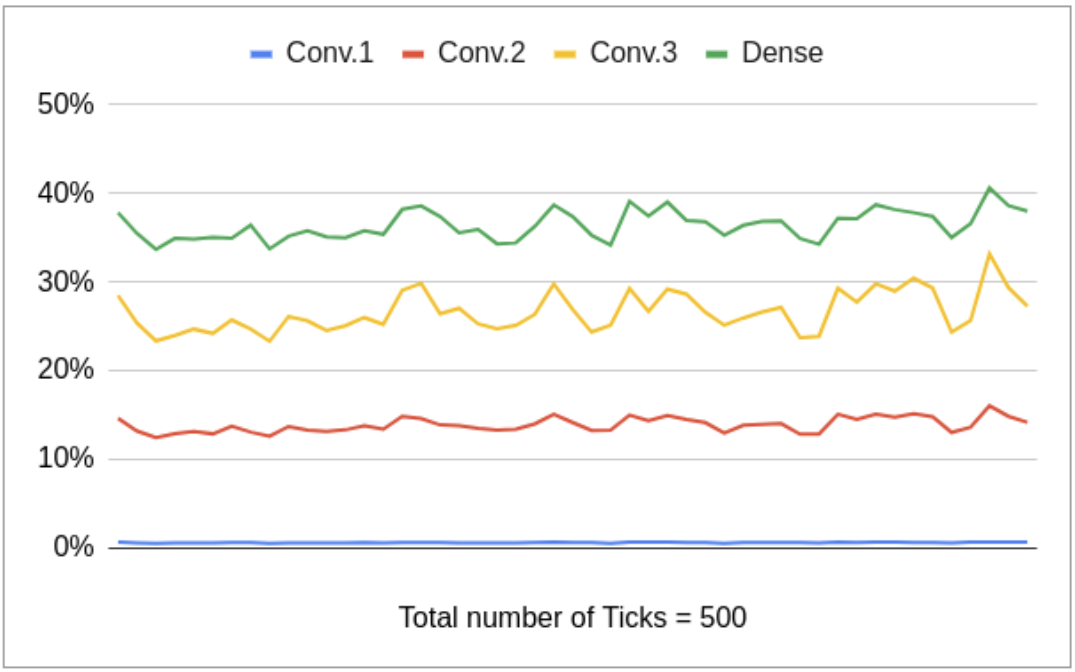


Figure 2.16. The ratio of spiking neurons in input to each layer of S2N2_cv. Ratios are collected during classifying one input (500 ticks) with trained weights.

The high BRAM utilization in Table 2.9 is because of the memory required to store membrane potentials for the LIF model (Figure 2.7).

Table 2.8. Required memory for buffering input at each layer of S2N2_cv is compared with tick-batching (Equation 2.5).

Layer	#Ticks	Input Size	Maximum Spike Ratio	Buffer Size Tick-Batching	Buffer Size S2N2_cv1	Improvement
Conv.1	500	28×28	0.7%	5,2136 bits	49 bits	×1,064
Conv.2	500	16×13×13	16%	4,542,720 bits	784 bits	×5,794
Conv.2	500	24× 11 ×11	33%	10,062,360 bits	1,176 bits	×8,556
Dense	500	32×4×4	41%	1,889,280 bits	10 bits	×188,928

Table 2.9. Synthesis results for S2N2_cv network architecture.

Network	Board	BRAM_18K	DSP48E	FF	LUT	Tick Resolution
S2N2_cv	PYNQ	35%	¡1%	2%	6%	30 ns

2.5 Conclusion

In this work, we introduced a streaming accelerator for spiking neural networks, namely S2N2. We showed that in batch-ticking, the buffer size used for storing encoded events depends on the input's spike ratios. This method is used in previous work, assuming a low spike ratio in the input. We showed that this assumption could be a tight condition on input's spike ratio. We then described how a binary tensor could address events and confirmed that a binary tensor with our streaming architecture requires less memory in our example applications.

We also described how to efficiently support axonal and synaptic delays in a feedforward SNN with only interlayer connections. By using binary tensors as inputs, we built our architecture upon FINN platform. We provided support for the LIF model in FINN and optional initialization of membrane potentials for each input to support SNNs in FINN.

Our streaming SNN architecture is suitable for processing signals of large temporal

dimension. Two novel SNN architectures for AMC are introduced in this work. In addition, an example of image classification on a SNN is described. All example applications are evaluated with the Vivado-HLS™tool. Our results achieve a minimum tick-resolution of 30 ns. S2N2 reduces input buffers' memory utilization by more than three orders of magnitude.

This chapter, in full, is a reprint of the material as it appears in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays 2021. Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Filtering Noise in SNN Input

Vision sensors are biologically inspired event-based image sensors. Unlike ordinary image sensors they only produce events if they detect changes in light intensity. It enables them to have an efficient output stream by excluding redundant data and only including changes. In addition, their architecture allow each pixel to be sampled at very high frequencies, for example, DAVIS sensor is capable of sampling at 333.3 kHz per pixel [10, 67, 78].

Neuromorphic sensors have seen growing importance in industry and research [69, 103]. For example, Samsung recently announced that an event-based image sensor, the Dynamic Vision Sensor (DVS) [67] will be used in their products for gesture recognition [15] alongside IBM's TrueNorth processor [30].

Event-based image sensors are extremely sensitive to Background Activity (BA) noise produced by temporal noise and junction leakage currents [67, 72, 128]. BA noise happens when output of a pixel changes under constant illumination. This noise can be removed by spatiotemporal correlation filters [99].

The programmable logic (PL) at the sensor head can be used for implementing the filter. By having the spatiotemporal correlation filter at the sensor side, the BA events will not be sent to a host PC. It can improve both the sensor's bandwidth utilization and processing. Implementing a spatiotemporal filter at the sensor head becomes a must if the sensor's PL hosts an application [69].

However these filters have two main problems: I) $O(N^2)$ memory complexity that makes their hardware implementation challenging and II) Inability of passing all of the real events. To elaborate on the second issue, it happens when an earlier filtered event finds spatiotemporal correlation with a current event. This earlier event, now has support from a current event to pass the filter, but it requires the filter to have additional memory for keeping all the information for the earlier event. This additional memory will increase the memory complexity of the filter even more and requires bigger PLs.

In this work we address these two issues by introducing a novel hardware friendly spatiotemporal correlation filter with $O(N)$ memory complexity for reducing noise in neuromorphic vision sensors.

Dynamic Vision Sensor (DVS)

In this work we use the Dynamic Vision Sensor, DVS128 from INILabs [48] similar to what is used by IBM and Samsung. The DVS128 sensor is an event-based image sensor that generates asynchronous address events as soon as the changes in log intensity since the last event exceed an upper or lower threshold.

Each pixel independently and in continuous time quantizes local relative intensity changes to generate spike events. If changes in light intensity detected by a pixel since the last event exceed the upper threshold, pixel will generate an ON event and if these changes pass the lower threshold pixel will generate an OFF event. A Pixel will not generate an event otherwise.

By this mechanism, DVS128 only generates events if there is a change in light intensity, therefore, sensor's output stream only includes the detected changes in sensed signal and does not carry any redundant data.

DVS sensor produces two types of events, ON and OFF. These events are in the form of an address-event that are generated locally by the sensor, each ON or OFF event includes polarity, x-position, and y-position of a pixel's event. The timing information of these events is coded in a 32 bits time-stamp.

To encode all the event information for output stream, DVS sensor uses Address Event Representation (AER) protocol [78] to create a quadruplets for each event as following:

$$e(p,x,y,t) \tag{3.1}$$

- p: Polarity, direction of change in light intensity
- x: Column number.
- y: Row number.
- t: Time-Stamp.

Background Activity (BA)

Background Activity noise is produced by thermal noise and junction leakage currents acting on switches connected to floating nodes [67, 127, 128]. These events decay the quality of the data and utilize unnecessary communication bandwidth and processing.

The difference between BA events and the real activity events of a pixel is that the BA events lack temporal correlation with events in their spatial neighborhood unlike the real events that have a temporal correlation with events from their spatial neighbors. Using this difference, the BA noise can be filtered out by detecting events generated by a pixel without the spatiotemporal correlation with the events generated by neighboring pixels and the pixel itself.

Such a filter is a spatiotemporal correlation filter. To process an event, a spatiotemporal filter searches the event's spatial neighborhood for events with time-stamps closer than a dT to the processing event's time-stamp (Fig. 3.1). If there exists an event with a time-stamp closer than the dT to the processing event's time-stamp, the processing event has *support* and can pass the filter. The processing event will be filtered out otherwise. This principal can be formulated as following:

$$
\begin{aligned}
e(p,x,y,t)\ is\ not\ BA \iff \exists\ |t-t_{ij}| < dT \\
s.t.\ |i-x| \leq 1\ \wedge\ |j-y| \leq 1
\end{aligned}
\tag{3.2}
$$

In the above equation, e is the processing event and t_{ij} is the time-stamp of the most recent event at $col = i$ and $row = j$ excluding the processing event.

It should be clear that for implementing such a filter one memory cell per pixel is required to store the most recent time-stamp.

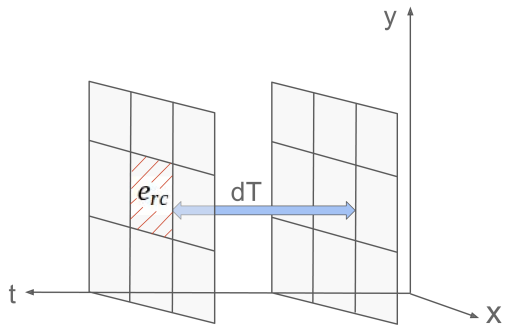


Figure 3.1. Principal of spatiotemporal correlation filter. An event can pass the filter if it has correlation with its spatial neighbors within a temporal window dT.

Contribution

We introduce a novel filter with $O(N)$ memory complexity for reducing BA noise in neuromorphic sensors. Our filter's memory requirement is significantly lower than other related work; this low memory requirement makes our filter desirable for near sensor implementation. By design, our filter stores all the necessary data for recovering recent events. By recovering past real events, we improved the filter's output up to 180% compared to other designs. We also improved the error rates of hardware friendly spatiotemporal filters. Error rates for our filter are 100× smaller than other hardware friendly designs for *false negative* error and zero for *false positive* error.

The rest of the chapter is organized as follows. Section 3.1 reviews the related work

in spatiotemporal correlation filter design. Section 3.2 describes the design of our proposed spatiotemporal filter. Section 3.3 studies BA noise and provides a mathematical model for it in neuromorphic sensors. In section 3.4 we define three types of error for spatiotemporal filters and compare them for different filter designs. Section 3.5 compares and describes the results of hardware implementation for different filter designs.

3.1 Related Work

Filtering BA noise at the sensor head improves the quality of the data at sensor's output stream, the bandwidth utilization, and saves on processing at a consumer of the data. Filtering BA noise at the sensor head can be inevitable if the existing PL at the sensor be used for implementing a custom application [69].

However implementing a spatiotemporal filter at the sensor head can be problematic. These filters require N^2 memory cells for a sensor with $N \times N$ pixels. Even for small filters, this memory requirement can exceed the available hardware resources at the sensor head. Even if a spatiotemporal filter fit into the sensor's PL, there will not be enough space left for implementing other applications and near sensor processing.

Liu, et. al. [72] designed a filter to address this issue by sub-sampling pixels into groups and projecting each group into one memory cell. An $N \times N$ sensor then will be divided into N^2/S^2 groups with S being the sub-sampling factor. Although this filter does not have significant loss in accuracy for $S = 2$, its error rate increases significantly as the sub-sampling factor grows.

The other problem with *Liu's* filter is the fact that pixels are only compared with other pixels in their sub-sampling group; if a real signal maps on different neighboring sub-sampling groups, *Liu's* filter will not search neighboring pixels in other sub-sampling groups for temporal correlation and it can increase the error rate (Fig. 3.2). This error can be significant when a DVS is used for observing small objects with limited movement [22].

As mentioned before, the high memory requirement of spatiotemporal filters drives a

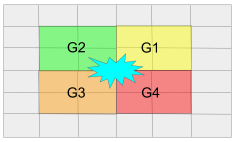


Figure 3.2. Sub-sampling groups G1, G2, G3, and G4 each includes 4 pixels. A real world signal that is mapped to different sub-sampling groups may not completely pass *Liu's* filter.

secondary issue in their design and prevents them to pass both supporting events, meaning if an earlier event that did not pass the filter, provides support for a current event, now the current event is also supporting the earlier event and both events should pass the filter. But this requires extra memory to store all the information for the earlier event. *Liu's* filter also lacks this feature and is not designed to store previous events' (x, y) positions and polarity.

3.2 Proposed Spatiotemporal Filter

In order to search for correlation, spatiotemporal filters need to store the time-stamp of earlier events. But compared to real signals, BA noise is a sparse and random signal. Our observations and calculations show that it is possible to take advantage of this property and store less time-stamps in a specific way to create an accurate and compact filter. In our filter, instead of just saving the time-stamp, we store all the information for an event. By storing all the event information, we are able to search for spatiotemporal correlation in future time of an event and recover all of the real events.

In our approach, Instead of using one memory cell per pixel to store the most recent time-stamp or in *Liu's* case, using one memory cell per sub-sampling group, we assign two memory cells to each row and each column to store the most recent event in the entire row or column (Fig. 3.3).

Each memory cell is 32 bits, to store the data for the most recent event, we use two memory cells: one for storing the time-stamp and one memory cell for storing a bookkeeping bit, the other axis position, and polarity (Fig. 3.4.a).

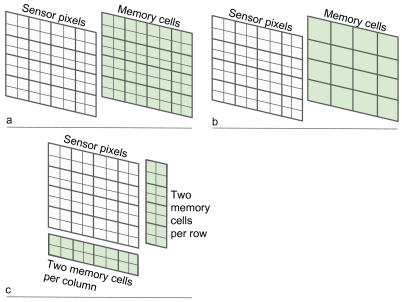


Figure 3.3. Memory utilization for different spatiotemporal filter designs: a) Baseline design: using one memory cell per pixel, b) *Liu's* design: using one memory cell per sub-sampling group ($S = 2$), c) Proposed design: two memory cells assigned to each row and column.

The bookkeeping bit is used for keeping a record of the stored event's status to prevent sending duplicate events. To store the event's polarity one bit is required, it leaves 30 bits from the memory cell for storing the other axis position, it allows the filter to support sensors as big as $2^{30} \times 2^{30}$ pixels.

For example after the filter finishes processing an incoming event $e = (p, x, y, t)$, it updates the cells corresponding to $row = y$ and $col = x$, both time-stamps will be updated to t and both polarity bits will be updated to p the value of *other axis position* for $row = y$ will be updated to x and the value of *other axis position* for $col = x$ will be updated to y. And if the result of processing is that the event is passing the filter, the bookkeeping bit will be set to one and zero otherwise. (Fig. 3.4.b).

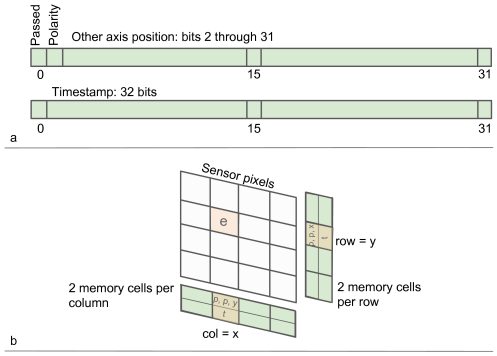


Figure 3.4. Memory utilization: a) Two memory cells per row and column, 1 bit to keep track if the event is passed, 1 bit to store polarity, 30 bits to store other axis position, and 32 bits to store time-stamp b) An arriving event $e(p, x, y, t)$ is stored in corresponding memory cells.

Using only two memory cells per row or column, significantly reduces the memory requirements. Compared to other designs, the memory complexity is reduced from $O(N^2)$ to $O(N)$. It makes our filter a much more affordable design for hardware implementation.

Because this filter is able to store all the information for an event, it can recover more real events, later in section 3.4 we show that this technique improves the data density of real events by about 180%.

3.3 Noise Model

In a CMOS image sensor, temporal noise is primarily due to the photodetector shot noise, the output amplifier's thermal and $1/f$ noise, and pixel reset, follower, and access transistor thermal, shot, and $1/f$ noise [127, 128]. Hand analysis of the CMOS image sensors published by several authors [10, 20, 67, 84, 136] show that at low illumination the dominant source of noise is reset and readout transistors, while at high illumination the dominant source of noise is the photodiode shot noise.

The DVS sensor is an unconventional CMOS imager. In this sensor, a pixel generates an output if there is enough change in the light intensity since the last event. A pixel then uses two comparators to generate a single bit for reporting an increase or a decrease in the light intensity and sensor generates an ON or OFF event accordingly.

In neuromorphic vision sensors BA events are produced under constant illumination. These events are caused by thermal noise and junction leakage currents acting on switches connected to floating nodes [10, 67, 128]. The hand analysis of DVS128 show that this sensor produces BA events with an average rate of 0.05 Hz at room temperature and it increases to 1.5 Hz at 60 °C [67].

These events randomly appear in time independent of each other with an average rate. Although their source is a combination of different noises (Shot, Gaussian, and Pink noises), we assume that their appearance follow a Poisson distribution. Our motivation for making this

assumption is based on our observations and previous related studies [127, 137].

To evaluate our assumption, we collected the output stream for a DVS128 sensor in a controlled environment. In our experiments by isolating the setup, we ensured that there are no real events captured by the sensor. We collected sensor's output stream for intervals of 7200 sec in three different constant illumination settings: dark, normal, and bright. We repeated each test for seven days (Fig. 3.5). We then used the collected data to measure the goodness of fit between their underlying distribution and Poisson distribution using Kolmogorov-Smirnov test [28].

Figure 3.5. DVS128 sensor used in our experiment for data collection. In each data collection experiment, the sensor was kept under constant illumination in an empty room isolated from any activity.

Poisson Distribution

Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time if these events occur with a known average rate and independent of each other:

$$P\{n\ events\} = \frac{(\lambda)^n}{n!}e^{-\lambda} \tag{3.3}$$

with λ being the average rate of occurrence for the events.

Two-Sample Kolmogorov-Smirnov Test

This test is one of the most popular and important tests for comparing samples with a reference probability distribution [100] and can serve as a goodness of fit test.

If *null hypothesis*, be the position that *"there is no relationship between two measured phenomena"*, Kolmogorov-Smirnov test can check the goodness of fit between samples drawn from an unknown distribution and samples drawn from a known distribution by rejecting or accepting the null hypothesis.

$$D_{m,n} = \sup_x |F(x)_m - F_n(x)| \tag{3.4}$$

Where $D_{m,n}$ is the Kolmogorov-Smirnov statistic, $\sup_x$ is the supremum of the set of distances, $F_m(x)$ is the cumulative distribution function of the known distribution, and $F_n(x)$ is the empirical distribution function (EDF) for n samples and is defined as:

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} 1_{x_i \leq x} \tag{3.5}$$

The null hypothesis is rejected at level α if:

$$D_{m,n} > \sqrt{-\frac{m+n}{2mn} ln(\frac{\alpha}{2})} \tag{3.6}$$

Applying the Kolmogorov-Smirnov test on the collected noise from our DVS sensor results an average pValue= 0.97 and KS statistic= 0.02 that confirms that the null hypothesis can be rejected at level $\alpha = 0.05$ between the BA events collected from the DVS sensor and Poisson process with average BA rate equal to 0.05 Hz (Fig. 3.6).

Our tests confirm that the BA events from the DVS can be assumed to be drawn from a Poisson distribution, to calculate the number of arrivals for any finite time interval, Poisson process can be used:

$$P\{N(t) = n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{3.7}$$

In (3.7), $P\{N(t) = n\}$ is the probability of receiving n BA events during time t from one pixel and λ is the average rate of BA noise-events per pixel.

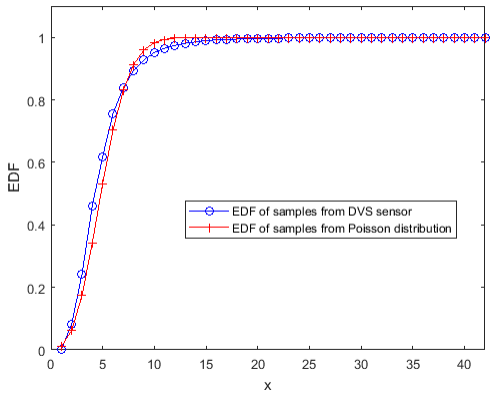


Figure 3.6. Kolmogorov-Smirnov test results for DVS128 BA noise and Poisson distribution: pValue = 0.97, KSstatistic = 0.02

3.4 Filters' Error Analysis

In this section, we use the noise model introduced in last section in our analysis to calculate the probability of error for different filter designs. We define three cases of *error* for spatiotemporal filters in our analysis:

- *False positive*: passing an event with no correlation with neighboring events.
- *False negative*: filtering an event with correlation with neighboring events.
- *Past event false negative*: filtering an event with correlation with neighboring events in future time.

Reader should note that if a BA event has correlation with real events it will pass any filter working on spatiotemporal correlation principals.

3.4.1 Baseline BA Filter

In this filter one memory cell is assigned to each pixel for storing the last event's time-stamp. This design, does not have *false positive* and *false negative* errors. However, this filter does not have enough memory to store other parameters of previous events, such as polarity therefore it is prone to *past event false negative* error. We use this design as our baseline.

3.4.2 Liu's BA Filter

This filter uses sub-sampling groups to reduce the memory size. Each sub-sampling group of pixels with sampling factor S, includes S^2 pixels and uses one memory cell for storing the time-stamp of the most recent event of the group (Fig. 3.3.b).

Grouping pixels in sub-sampling groups bigger than 2×2 causes false spatiotemporal correlation between non-neighboring pixels and will produce *false positive* error.

Although this filter is more efficient than the *baseline* filter for utilizing memory cells, it still does not store any data related to events beside their time-stamp. As a result, this filter is incapable of recovering past events with spatiotemporal correlation with current events and is prone to *past event false negative* error.

False negative error in this filter is caused by its specific design. This filter does not check neighboring groups for supporting an arrival event. In the case of having a real world signal mapped to neighboring pixels in different sub-sampling groups (Fig. 3.2), this filter may not pass all of the bordering events.

To calculate the probability of error for an incoming event in this filter, we define three pixel groups for a sub-sampling group and calculate their *false positive* and *false negative* probabilities.

A S^2 sub-sampling group of pixels includes: 4 corner pixels, $4(S-2)$ side pixels, and $(S-2)^2$ inner pixels (Fig. 3.7).

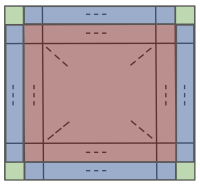


Figure 3.7. Three pixel groups for a $S \times S$ sub-sampling group: *green*: 4 corner pixels, *blue*: $4(S-2)$ side pixels, and *red*: $(S-2)^2$ inner pixels.

$$S^2 = 4 + 4(S-2) + (S-2)^2 \tag{3.8}$$

An arriving event can be from corner, side or inner groups with the probabilities $4/S^2$, $4(S-2)/S^2$, and $(S-2)^2/S^2$ accordingly.

Each corner pixel has five neighboring pixels outside of its sub-sampling group, that can cause *false negative* errors. And has S^2-4 non-neighboring pixels in its sub-sampling group that can cause *false positive* errors.

Each side pixel has three neighboring pixels outside of its sub-sampling group that can potentially cause *false negative* errors. And has S^2-6 non-neighboring pixels in its group with potentials of causing *false positive* errors. For the last group, each inner pixel has S^2-9 non-neighboring pixels that can cause *false positive* errors.

In *Liu's* filter *false positive* error is when a non-neighboring pixel's spatiotemporal correlation with an arriving event is used to pass the event and can be calculated for a temporal window t as:

$$\begin{aligned} P\{false\ positive\ error(t)\} = \\ \frac{4}{S^2}(1-P\{N(t)=0\}^{S^2-4}) \\ +\frac{4(S-2)}{S^2}(1-P\{N(t)=0\}^{S^2-6}) \\ +\frac{(S-2)^2}{S^2}(1-P\{N(t)=0\}^{S^2-9}) \end{aligned} \tag{3.9}$$

$P\{N(t)=0\}$ is calculated using (3.7) and $1-P\{N(t)=0\}^k$ is the probability of having at least one BA event from k pixels during time t.

To calculate the *false negative* error for *Liu's* filter, we calculate the possibility of loosing support from a neighboring pixel in a different sub-sampling group:

$$P\{false\ negative\ error\} = \frac{4}{S^2}(\frac{5}{9}) + \frac{4(S-2)}{S^2}(\frac{3}{9}) \tag{3.10}$$

3.4.3 Normal Sub-Sampling Filter

To resolve the *false negative* error in *Liu's* filter shown in Fig. 3.2, we consider a filter that neighboring sub-sampling groups can support each other. To the best of our knowledge this filter is not used in practice and is provided only to demonstrate that resolving the *false negative* error will increase the *false positive* error in sub-sampling approach. In the rest of this chapter, we refer to this filter by *sub-sampling filter*.

This filter is also prone to *past event false negative* error, but because neighboring sub-sampling groups in this filter can support each other the *false negative* error is zero. To calculate the *false positive* error we need to calculate how many non-neighboring pixels can cause this error for each pixel in a sub-sampling group. For the corner pixels, there are $4S^2-9$ non-neighboring pixels that can potentially cause this error. For side and inner pixels, there are $2S^2-9$ and S^2-9 pixels accordingly that can potentially cause *false positive* error. We can formulate this for a temporal window t as:

$$\begin{aligned} P\{false\ positive\ error(t)\} = \\ \frac{4}{S^2}(1-P\{N(t)=0\}^{4S^2-9}) \\ +\frac{4(S-2)}{S^2}(1-P\{N(t)=0\}^{2S^2-9}) \\ +\frac{(S-2)^2}{S^2}(1-P\{N(t)=0\}^{S^2-9}) \end{aligned} \tag{3.11}$$

3.4.4 Our Proposed Filter

This filter stores the quadruplet (3.1) of the most recent event in a row or a column. Therefore it is capable of recovering previous events and does not suffer from *past event false negative* error.

To store an event, this filter stores both row and column information, therefore non-neighboring events are not included in the search for spatiotemporal correlation. As a result the

false positive error in this filter is zero.

The *false negative* error happens in a special case when there is only one real event to provide support for another real event but the data of the older real event gets replaced by BA noise. Let's consider two real events e_1 and e_4 in neighboring of each other with the timing order of $t_1 < t_4$:

$$\begin{aligned} &e_1(p_1,x_1,y_1,t_1) \textit{ and } e_4(p_4,x_4,y_4,t_4) \textit{ are neighbors} \\ &\iff |x_1 - x_4| \leq 1 \wedge |y_1 - y_4| \leq 1 \end{aligned} \tag{3.12}$$

If $t_4 - t_1 < dT$ then both events must pass the filter, but if BA events overwrite e_1's data, the recent event e_4 will lose its support from e_1. If noise does not support e_4 (no BA event in e_4's neighborhood during dT) e_4 will not pass the filter and results into *false negative* error. Let's consider two BA events $BA_2(p_2,x_1,y_2,t_2)$ and $BA_3(p_3,x_3,y_1,t_3)$ (Fig. 3.8), *false negative* error will happen if:

$$t_1 \leq (t_2,t_3) \leq t4 \ \wedge \ |y_2 - y_4| > 1 \ \wedge \ |x_3 - x_4| > 1 \tag{3.13}$$

Our filter's *False negative* error for a $M \times M$ sensor and temporal window t can be calculated as:

$$\begin{aligned} &P\{false\ negative\ error(t)\} = \\ &(1 - P\{N(t) = 0\}^{M-3})^2 \end{aligned} \tag{3.14}$$

In (3.14) we are calculating the probability of receiving at least one BA event from the pixels in e_1's row (y_1) excluding e_2's neighbors and at least one BA event from the pixels in e_1's column (x_1) excluding e_2's neighbors during a temporal window t.

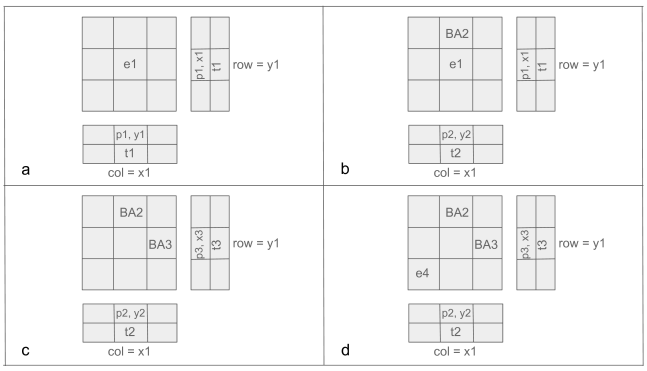


Figure 3.8. Example of *false negative* error in our proposed filter: a) at $t = t_1$, real event e_1 arrives, column x_1 stores y_1 and t_1 and row y_1 stores x_1 and t_1. b) at $t = t_2$, noise event BA_2 arrives and changes the values of column x_1 to y_2 and t_2. c) at $t = t_3$, noise event BA_3 arrives and changes the values of row y_1 to x_3 and t_3. At this point information related to e_1 are completely overwritten by noise events. d) at $t = t_4$ real event e_4 arrives and filter can not find a temporal correlation in its neighboring pixels and *false negative* error occurs.

3.4.5 Theoretical Comparison

To compare the filters, we use our developed equations to calculate both *false negative* and *false positive* errors.

In our comparison for *false positive* error, we set the temporal window of the filter to a practical value $dT = 1$mSec and noise frequency to 0.05 Hz for room temperature [67]. This error is zero in our filter and the result for other sensors with different sensor sizes are shown in Fig. 3.9.

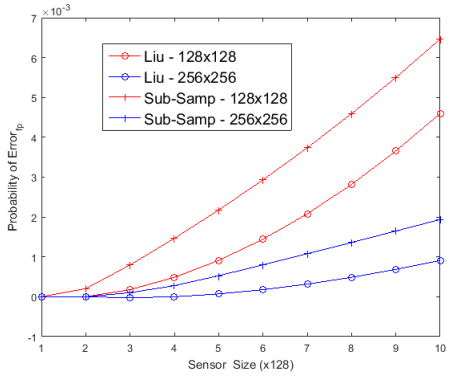


Figure 3.9. False positive error calculated for $dT = 1$mSec and $f_{noise} = 0.05$Hz (room temperature). This error is zero for our filter. Increasing the size of the filter for *Liu's* and *sub-sampling* filters results in smaller sub-sampling groups and improves this error.

Increase in temperature increases the average noise rate and results to higher *false positive* error rates for *Liu's* and *sub-sampling* filters.

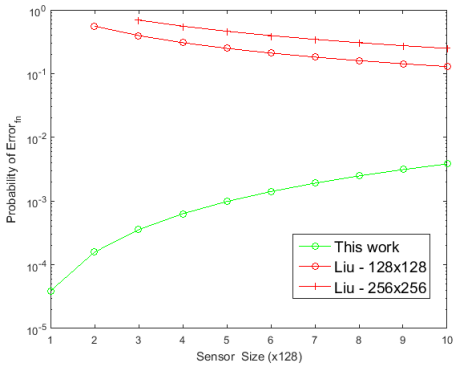


Figure 3.10. False negative error calculated for temporal window $dT = 1$mSec and noise rate $f_{noise} = 0.05$Hz. Since this error is significantly lower for our filter, data are presented in logarithmic scale. Increasing *Liu's* filter's size will increase the number of corner and side pixels and results in higher error rates (3.10).

Comparison between the filters for *false negative* error is done in a different fashion. This error is zero for *sub-sampling* filter, temporal independent for *Liu's* filter (3.10), and temporal dependent for our proposed filter (3.14).

Fig. 3.10 shows this comparison between *Liu's* and our filter. The decay in *Liu's* noise probability is because of fading effect of corner and side pixels for larger sub-sampling groups (3.10). Increasing the temporal window will increase the probability of *false negative* error for our filter.

3.4.6 Comparison Between Filters using Real Data

In this subsection, we compare the filters using real data captured from a DVS sensor. To compare the performance of our filter with the baseline filter, we passed the collected noise from section 3.3 to both filters. Our results show that our filter works as expected and we did not observed any error during our observations (Fig. 3.11).

To compare the filters for *past event false positive* error, we collected the output of the DVS128 sensor while moving a laser pointer in front of the image sensor. We collected this data

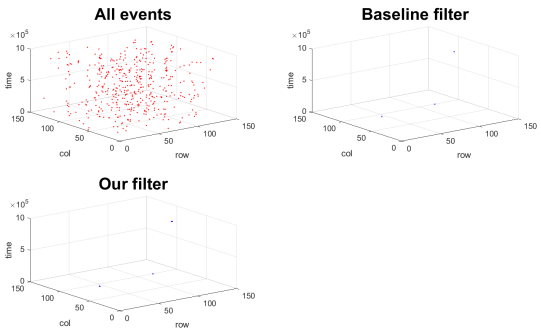


Figure 3.11. Comparison between our filter and baseline filter using real data captured by a DVS sensor. Both filters identically remove the BA noise.

for two seconds and used all four filters for denoising. We repeated this test for 20 times and calculated the number of passed events between the filters. We concluded that on average our proposed filter passes 180% more real events compared to other filters. The result of one of our tries is shown in Fig. 3.12.

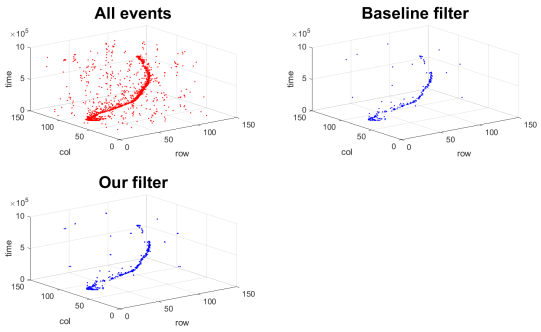


Figure 3.12. *Past event false negative* noise. Data captured with a moving laser pointer in front of the camera. Compared to other filters, our proposed filter is able to pass 180% more real events compared to other filters. All other filters are prone to this error and output of the baseline filter can also represent *Liu's* and *sub-sampling* filters for this error.

3.5 Hardware Implementation

To compare the resource utilization between our proposed filter and other approaches, we implemented all the filters using Vivado ® High Level Synthesis.

Baseline filter's size is equal to the sensor size and *sub-sampling* and *Liu's* filters' sizes

can be different depending on their sub-sampling factor (Fig. 3.3). But for an equal filter size, hardware utilization for these three filters are almost identical. Therefore a reader can assume that the provided result for a *baseline* filter are valid for the same size *Liu's* or *sub-sampling* filters.

In practice, because of limited real estate and to conserve energy and heat, compact FPGAs with high performance-per-watt ratios are used at the sensor head. Therefore we used Artix®-7 from Xilinx® for our synthesis.

The result of synthesis for different sensor sizes range from 128×128 to 1280×1280 is shown in Table 3.1.

Table 3.1. Comparison between resource utilization

Size	filter	Latency (nSec)	BRAM	FF	LUT	Throughput (MHz)
128×128	baseline	8	8.77%	0.06%	0.32%	14
	this work	35	0.55%	0.31%	0.72%	3
256×256	baseline	8	35.07%	0.06%	0.34%	14
	this work	35	0.55%	0.32%	0.72%	3
512×512	baseline	8	140.27%	0.07%	0.37%	14
	this work	35	0.55%	0.32%	0.72%	3
1024×1024	baseline	8	561.10%	0.10%	0.40%	14
	this work	35	1.10%	0.32%	0.73%	3
1280×1280	baseline	8	1122.19%	0.12%	0.43%	14
	this work	35	2.19%	0.32%	0.73%	3

Table 3.1 shows that for sensor sizes bigger than 256×256, *baseline* filter can not fit in the fabric. And since *Liu's* and *sub-sampling* filters have similar hardware utilization as *baseline* filter for equal filter size. These two filters also can not fit in the fabric if they are bigger than 256×256 (Fig. 3.13).

As it is shown in Table 3.1, our proposed filter consumes less memory and as a trade off, it has lower throughput and higher latency compared to baseline filters. However its latency is three orders of magnitude faster than the sensor, pixels of DVS128 have a latency equal to 15 µsec which is equal to 66.7 kHz and the sensor itself is capable of producing maximum 1M events per second.

Compared to other filters, our proposed filter does not have a high demand on resources.

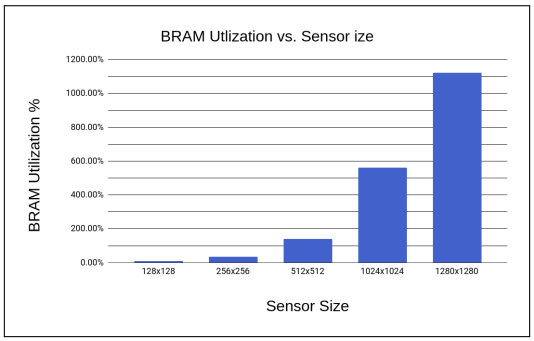


Figure 3.13. Memory utilization for baseline filter. Liu's and sub-sampling filters with sizes equal to baseline filter have similar memory utilization.

Even for the large sizes it utilizes a fraction of memory compared to other filters. Fig. 3.14 shows a comparison between the smallest *Liu's* filter and our proposed filter for different sensor sizes.

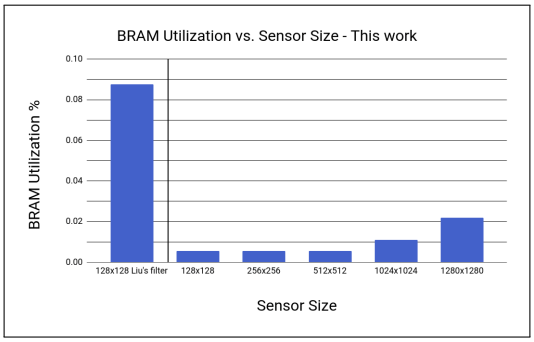


Figure 3.14. Memory utilization for proposed filter. Even for large sensor sizes, its memory utilization is significantly lower than a 128×128 Liu's, baseline, or sub-sampling filter.

3.6 Conclusion

This chapter presents a novel $O(N)$ spatiotemporal filter for neuromorphic vision sensors. By modeling the noise of neuromorphic vision sensors, we calculated the probability of error for our proposed sensor and other related filter designs. Our error models show that the proposed filter has $100\times$ less *false negative* error compared to other hardware friendly designs and zero *false positive* error. By collecting data from a real sensor, DVS128 we showed that the performance of our proposed filter follows our predictions and developed models. In addition, this filter shows an improved output up to 180% compared to all other designs by passing all of

the real events.

In our hardware implementation section we showed that this novel filter reduces the memory utilization by $10\times$ and can fit on fabrics with limited resources and unlike other spatiotemporal filters, it still leaves enough space on the fabric for implementing other possible applications.

This chapter, in full, is a reprint of the material as it appears in IEEE Transactions onEmerging Topics in Computing 2017. Alireza Khodamoradi and Ryan Kastner. The dissertationauthor was the primary investigator and author of this paper.

Chapter 4

Reshaping Residual Neural Networks

Convolutional Neural Networks (CNNs) [59, 61] have shown remarkable breakthroughs for image classification [113, 142]. These networks inherit their name from integrating multiple convolutional layers in their architecture. In these classifiers, different levels of features (low, medium, and high) can be enriched by increasing the number of stacked layers (*depth*) [117, 123, 142].

Increasing the network depth requires more complex training methods. For example, training networks with tens of layers only became possible with normalized initialization [39, 62] and batch-normalization [49]. However, simply stacking more layers to create deeper networks results in no improvement (saturation) or degradation in accuracy [37, 121].

Residual Networks (ResNets) [38] address this problem using skip connections that connect the output of a layer to a post-nonadjacent layer's input (Figure 4.1.a). These shortcuts between the layers increase the accuracy for networks with tens of layers and make it possible to train networks with hundreds or even thousands of layers.

Skip connections between nonadjacent layers in ResNets introduce an architectural irregularity compared to previous CNNs such as AlexNet [59] and VGG [117]. Therefore accelerating a ResNet's inference stage for real-time performance on resource-limited devices such as FPGAs has become a challenging task. This complication is essentially caused by the data's lifetime in the skip connection path (Figure 4.1.a). The skip connection input is also the

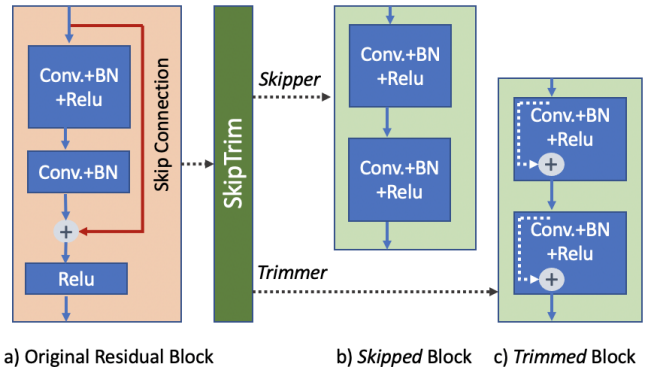


Figure 4.1. a) Long skip connection used in ResNets. b) for smaller ResNets, SKIPTRIM prunes away all the skip connections (*SKIPper*). c) for larger ResNets, SKIPTRIM breaks a long skip connection into multiple shorter skip connections (*TRIMmer*).

input to a second path parallel to the skip connection. The second path usually contains multiple convolutional, batch-normalization, and activation layers. In contrast, the skip connection has no layers (or only a single convolutional layer for scaling). Hence, because these two paths' outputs have to be added together, the skip connection will degrade the pipelining by holding on to its output until all the second path layers are finished processing. This stalling can be avoided by increasing the buffer capacity, either on-chip, leading to more on-chip memory utilization [6], or off-chip, increasing the off-chip memory bandwidth. Additionally, this requires extra control logic for scheduling [75, 76].

In an ideal case, the skip connections could be removed to simplify the hardware implementation. But eliminating skip connections either before or after training immensely reduces the inference accuracy [106, 140]. Skip connections are crucial for training larger networks. Removing the skip connections induces drastic changes to the architecture, but, as we show in this work, it is possible to remove or modify skip connections and achieve substantial resource reductions while incurring minimal or no loss in accuracy.

We introduce the SKIPTRIM, an iterative learning method that gradually modifies a ResNet architecture to temper the shock of taking away skip connections. Our approach slowly prunes away skip connections (Figure 4.1.b) or splits the skip connection into two shorter ones (Figure 4.1.c). These two optimizations allow SKIPTRIM to reduce the complexity of the ResNet

by modifying its skip connections while maintaining accuracy.

SKIPTRIM starts with a pre-trained ResNet with original skip connections. At each iteration, SKIPTRIM iteratively modifies the network considering performing local changes to the skip connections by removing or trimming them. In the end, all the original skip connections are either pruned away or split into shorter skip connections.

SKIPTRIM provides an automated technique to modify ResNet models to increase their hardware efficiency by removing or relaxing the implementation complexity of the skip connections. We show the value and applicability of these techniques on ResNets ranging from 20 to 110 layers trained on the CIFAR-10 [58], CIFAR-100 [58], SVHN [91], and RadioML.2018 [95] datasets, targeting FPGAs.

To evaluate the impact of ResNet skip connection removal in hardware, we synthesize both the original and trimmed ResNets for FPGAs using the hls4ml library [1,25,31] in Vivado™ HLS. Our results demonstrate that *SkipTrimmed* ResNets use 20% fewer BRAMs with no loss in accuracy for smaller models and 16% fewer BRAMs with less than 1% loss in accuracy for larger models.

Contributions:

- the SKIPTRIM method of gradually splitting or pruning away skip connections from ResNets with minimal to no loss in accuracy,
- splitting the skip connections in residual networks for more efficient FPGA implementations,
- and demonstrating that resulting networks are implemented more efficiently than the original ResNets in FPGAs

chapter Layout: In Section 4.1 we explain why skip connections are essential to ResNets, review previous works for realizing ResNets on FPGAs, and related training methods for eliminating skip connections. In Section 4.2, we explain the new training method used in

SKIPTRIM to relax the skip connections' implementation complexity. Section 4.3 introduces optimization for implementing short skip connections on FPGAs and provides our training results, including an example for a quantized model. Section 4.4 describes the limitations of our work, and conclusions are made in Section 4.5.

4.1 Background

4.1.1 Importance of Skip Connections

ResNets add skip connections to help mitigate the vanishing gradient problem when training deep neural networks. Training methods based on Stochastic Gradient Descent (SGD) [55, 104] optimize the network's objective function by adjusting its trainable parameters (e.g., weights) via backpropagation. Backpropagation reuses partial computations of the gradient from one layer in the gradient's computation for the previous layer [107], i.e., the chain rule. As more layers are added to the network, the partial gradient computation may lead to extremely small (vanishing) values for the layers far back in the backpropagation path, which is a problem for training deep neural networks [7, 34].

Residual networks [38] address the vanishing gradient using skip connections that connect one layer's output to a nonadjacent layer's input. These connections provide a direct path for propagating the error through the layers and dealing with the vanishing gradient problem. In the forward path, skip connections provide an identity mapping of their input to their output, which is essential for avoiding saturation in deep neural networks training [38, 40]. This identity mapping also enables the ResNets to have fewer filters/weights and lower complexity than other deep CNN architectures without skip connections [38], such as VGGs [117].

4.1.2 Accelerating ResNet Inference on Custom Platforms

Although skip connections make it easier to train deep neural networks, they introduce additional complexity for the hardware implementation. Realizing one skip connection requires

an extra addition and one optional scaling stage (implemented as a convolutional layer or padding) plus memory for buffering its data.

Buffering skip connection data can be done by using on-chip or off-chip memory. Implementing the skip connection on-chip increases the on-chip memory utilization [6]. Our experiments show that using skip connections, on average, results in a 20% increase in on-chip memory utilization. For larger networks, off-chip memory is used for buffering interlayer data. In this case, Deep Learning Accelerators (DLA) [66] load the input for each group of layers (convolution+batch-norm+activation) and return the output to off-chip memory. Scheduling can be controlled from the processing system (CPU) [26, 66], and with additional control logic, the DLA can be instructed to perform element-wise addition for realizing the skip connection [68, 75, 76, 145].

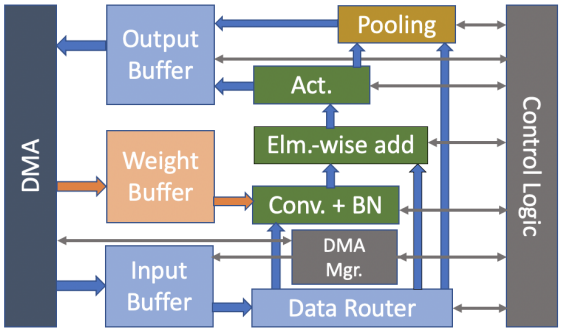


Figure 4.2. Deep Learning Accelerator (DLA) with long skip connections proposed in [75, 76], which requires an additional load for the skip connection data.

This will increase the bandwidth utilization for the off-chip memory. For example, suppose a skip connection hops over two residual layers (similar to Figure 4.1.a). In that case, for the first residual layer, the DLA in Figure 4.2 is instructed to process its data without the skip connection, and for the second residual layer, it will be instructed to load the skip connection's data for element-wise addition. This extra bandwidth utilization can be estimated by calculating the size of the input for each skip connection.

4.1.3 Removing Skip Connections

Removing skip connections from ResNets has been studied before. In [106] the authors introduce a new training method that starts with a standard ResNet. During the early stages of training, skip connections exist in the network. The training method includes an objective function that penalizes the skip connections by a Lagrange multiplier and causes them to subsequently phase out by the end of the training. This technique can remove the skip connections from smaller ResNets (18 to 34 layers) with a small decrease in accuracy between 0.5% and 3%.

DiracNets [140] replace the skip connections with Dirac parameterization, as shown in Equation 4.1.

$$\text{DiracNet [140]: } y = \sigma(x + Wx) \tag{4.1}$$

$$\text{ResNet [38]: } y = x + \sigma(Wx) \tag{4.2}$$

In the above equations, sigma is the activation function. For a better comparison with ResNets, Equation 4.2 is simplified to show only one convolutional layer. In fact, skip connections in ResNets hop over more than one convolutional layer. While in DiracNets, the identity mapping is over one single convolutional layer. Therefore the weights and the identity mapping of the input can be folded as following:

$$x + Wx = (I + W)x$$

This change requires DiracNets to use wider networks than the original ResNets. The authors showed that their technique could be used to create models with up to 34 layers. Although it works for smaller models, compared to ResNets with similar depths, DiracNets show a decrease in accuracy between 0.5% and 1.5% for different depths and datasets.

4.2 SKIPTRIM

Our method includes two parts: *SKIPper* and *TRIMmer* (SKIPTRIM). Both parts use an iterative training method based on Knowledge Distillation (KD) to relax the implementation complexity of the skip connections in ResNets by decreasing the residual blocks' irregularity.

KD is the process of transferring knowledge from a large model to a smaller one. This method was introduced in [42] as a compression method for neural networks, and its different variations have shown impressive results for compressing neural models for various applications [85, 109, 125]. These compression methods distill the knowledge from an easy-to-train and large model (teacher) to train a smaller network (student) that can not be trained from scratch. The teacher model is already trained, and the student model is trained to learn the teacher's exact behavior by replicating its output.

Skipper prunes all the skip connections from smaller ResNets (in Section 4.3 we show that this pruning technique results in acceptable accuracy for ResNets with up to 42 layers). For larger ResNets, removing skip connections results in a significant loss in accuracy. Therefore we introduce the *Trimmer* for splitting the skip connections into multiple short skip connections per residual block. We also introduce an optimized implementation for the short skip connections with a trivial increase in BRAM utilization. Pseudo code of our algorithm is shown in Algorithm 1.

4.2.1 Skipper

The Skipper iteratively removes all the skip connections from a ResNet model. This method starts with two identical and pre-trained ResNets with original skip connections. One of these networks preserves its structure during the training, and we refer to this network as the teacher model. The other ResNet network is referred to as the student model. Throughout the training, Skipper removes the skip connections from the student network one at a time.

As described in [40], reordering the layers in a residual block can improve the training

Algorithm 1: SKIPTRIM METHOD

```
α = tune-able parameter
teacher = pre-trained resnetx
student = pre-trained resnetx
if splitting then
    SkipTrim=Trimmer
    set α
end
if pruning then
    SkipTrim=Skipper
    set α
end
current-skip = student model's first skip connection from the input side
for i in training epochs do
    if i mod α then
        SkipTrim(current-skip)
        current-skip = student model's next skip connection from the input side
    end
    train using the loss function in Equation (4.4)
end
save the student model
```

results. Our results also confirm that the residual block shown in Figure 4.3.d provides higher accuracy than the original residual block introduced in [38] (Figure 4.3.a). Therefore our pre-trained network used for both student and teacher models have residual blocks similar to Figure 4.3.d. Hence, pruning one skip connection is done by modifying the forward path of its residual block as shown in Listing 4.1 (line 9 of Algorithm 1 changes the *reshape* from its default value, False, to True).

After pruning all the skip connections, to reduce the implementation complexity, adjacent batch-norm and convolutional layers can be folded to one layer according to the following equation:

$$\begin{aligned} y &= W_{bn}(W_{conv}x + b_{conv}) + b_{bn} \\ &= W_{bn+conv}x + b_{bn+conv} \end{aligned} \tag{4.3}$$

Removing each skip connection from the student network modifies the network's structure and results in a drop in accuracy. To regain this loss, we use the teacher's output to enhance the student model's training for producing correct labels. For this purpose, we use a loss function similar to [42] as follows:

$$\mathscr{L} = (1-\beta)\mathscr{G}(s-l) + \beta\mathscr{H}(t-s) \tag{4.4}$$

Here, $\mathscr{G}$ and $\mathscr{H}$ are distance functions, s and t are student and teacher output vectors respectively, and l is the correct label vector. β is a tune-able parameter.

4.2.2 Trimmer

The Trimmer splits a long skip connection into multiple shorter skip connections. Like the Skipper, it uses an iterative training method and the loss function described in Equation 4.4

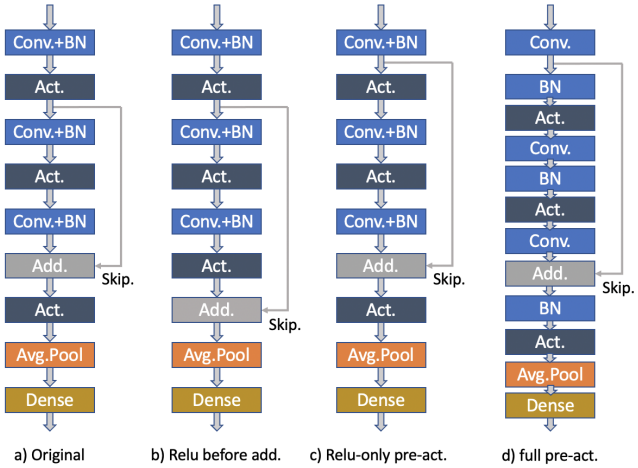


Figure 4.3. a) Residual block introduced in [38]. b, c, and d) Different versions of the Residual blocks introduced in [40].

to split the skip connections in the student model.

Different arrangements for a residual block are explained in [40] and displayed in Figure 4.3. Our goal for splitting a long skip connection into multiple short ones is to reduce the hardware implementation's complexity. In the following, we explain which of these arrangements is more suitable for this purpose.

Splitting the residual block shown in Figure 4.3.d into two shorter ones results in two (BN/Act./Conv.) blocks. In this case, because the BN and the Conv. layers are not adjacent, they can not be folded together as described in Equation 4.3.

Listing 4.1. pseudo code *Skipper*

```
if reshape:
    out = conv1(act(BN1(inp)))
    out = conv2(act(BN2(out)))
else:
    out = conv1(act(BN1(inp)))
    out = conv2(act(BN2(out)))
    out += inp
```

Splitting the arrangement shown in Figure 4.3.c results in two (Act./Conv.+BN). With the BN and the Conv. layers being adjacent, they can be folded together (Equation 4.3). This folding results in a smaller model size for Figure 4.3.c compared to Figure 4.3.d.

We can further improve upon (Act./Conv.+BN) by splitting Figure 4.3.b into two (Conv.+BN/Act.) blocks. This is because applying the activation after the convolution can be done at the loop that copies the convolution accumulators' results to the output [130]. This is not possible for (Act./Conv.+BN), and the activation requires a separate loop and one extra buffer [31].

The two arrangements shown in Figure 4.3.a and Figure 4.3.b are different only in one activation layer. Not including the activation layer in a residual block (Conv.+BN) is similar to Equation 4.1, and as shown in DiracNets, with wider layers, it results in less accuracy even for smaller ResNets [140].

Therefore, we select the (Conv.+BN/Act.) configuration for shorter skip connections, and line 5 of Algorithm 1 breaks a long skip connection into two shorter skip connections by changing the *reshape* from False to True in Listing 4.2.

Listing 4.2. pseudo code *Trimmer*

```
if reshape:
    out1 = act(BN1(conv1(inp)))
    out1 += inp
    out = act(BN2(conv2(out1)))
    out += out1
else:
    out = act(BN1(conv1(inp)))
    out = BN2(conv2(out1))
    out = act(out+inp)
```

In the next section we explain an optimization for realizing the short skip connections on FPGAs.

4.3 Experiments

We tested SkipTrim on ResNets with different depths from 20 to 110 layers against different datasets, including CIFAR10/100 [58], SVHN [91], and RadioML2018 [95]. In our setup, we trained our networks using Keras [14], mxnet [4], and Pytorch [98]. Our synthesized results are from Vivado™ HLS for Alveo U250 FPGA using code generated by hls4ml [31].

4.3.1 Training Results

As seen in Table 4.1, the skip connections can be removed from ResNets with up to 44 layers with improvements in accuracy. And for larger ResNets, removing the skip connections results in a significant loss in accuracy. Therefore, for larger ResNets, we split a large skip connection into two shorter ones.

Table 4.1. CIFAR10 Top1 accuracy results on different ResNet configurations

	Accuracy (%)		
Model	Original [38]	Teacher	SKIPTRIM
ResNet20	91.25	91.85 $\pm$ 0.06	92.2 $\pm$ 0.05 (Skip)
ResNet32	92.49	92.93 $\pm$ 0.07	93.02 $\pm$ 0.07 (Skip)
ResNet44	92.83	93.12 $\pm$ 0.09	93.21 $\pm$ 0.07 (Skip)
ResNet56	93.03	93.69 $\pm$ 0.06	92.72 (Skip) 93.25 $\pm$ 0.08 (Trim)
ResNet110	93.39 $\pm$ 0.16	94.31 $\pm$ 0.09	90.1 (Skip) 93.3 $\pm$ 0.08 (Trim)

Tables 4.2 and 4.3 show our results on CIFAR100 and SVHN datasets respectively. As seen in these tables, similar to Table 4.1, *Skipper* provides better results for larger networks.

Table 4.2. CIFAR100 Top1 accuracy results on different ResNet configurations

	Accuracy (%)		
Model	Original [111]	Teacher	SKIPTRIM
ResNet20	70.36	70.2 $\pm$ 0.08	70.3 $\pm$ 0.12 (Skip)
ResNet56	75.12	75 $\pm$ 0.11	71.1 (Skip) 75 $\pm$ 0.06 (Trim)
ResNet110	77.2	77.1 $\pm$ 0.15	68.2 (Skip) 76.9 $\pm$ 0.1 (Trim)

For RadioML.2018 dataset, we compared our work against [95] on ResNet34. In [129], the authors mention that realizing this network on FPGAs has high complexity due to the scheduling for the skip connections. Table 4.4 shows that SKIPTRIM can remove all the skip connections from this network with improved accuracy. With less BRAM utilization and no scheduling for the skip connections, this new network is primed for hardware implementation.

Table 4.3. SVHN Top1 accuracy results on different ResNet configurations

	Accuracy (%)		
Model	Original [111]	Teacher	SKIPTRIM
ResNet20	96.57	96.58 ± 0.09	96.68 ± 0.08 (Skip)
ResNet56	97.25	97.23 ± 0.11	94.4 (Skip) 97.33 ± 0.11 (Trim)
ResNet110	97.55	97.5 ± 0.15	85 (Skip) 97.56 ± 0.16 (Trim)

Table 4.4. RadioML.2018 top1 accuracy results on different ResBlock configurations

	Accuracy (%)		
Model	Original [95]	Teacher	SKIPTRIM
ResNet34	95.6	95.8 ± 0.14	96.6 ± 0.12 (Skip)

Training Parameters

In all of our training scripts, we set α in Algorithm 1 equal to one and three for *Skipper* and *Trimmer* respectively. For $\mathscr{G}$ and $\mathscr{H}$ in Equation 4.4, we used *cross_entropy* and *Mean Square Error (MSE)* respectively, and set β to 0.35. Accuracy results are average and standard deviation on three runs per experiment.

4.3.2 Optimizing Short Skip Connections on FPGAs

Table 4.5, compares the hardware utilization for different models shown in Figure 4.4. Pruning a skip connection (Figure 4.4.b) results in a 21% gain in BRAM utilization. Later in Section 4.3.3 we show that this percentage holds when multiple skip connections are pruned from a ResNet.

Table 4.5. Hardware utilization for networks in Figure 4.4

Model	BRAM	BRAM Normalized	DSP	FF	LUT
Fig. 4.4.a	508	1	1	129.8k	158.9k
Fig. 4.4.b	406	0.79	1	127.7k	163.7k
Fig. 4.4.c	556	1.1	1	131.9k	163.7k
Fig. 4.4.d	428	0.84	1	126.5k	150.9k

As described in the previous section, we selected the *Relu before addition* configuration

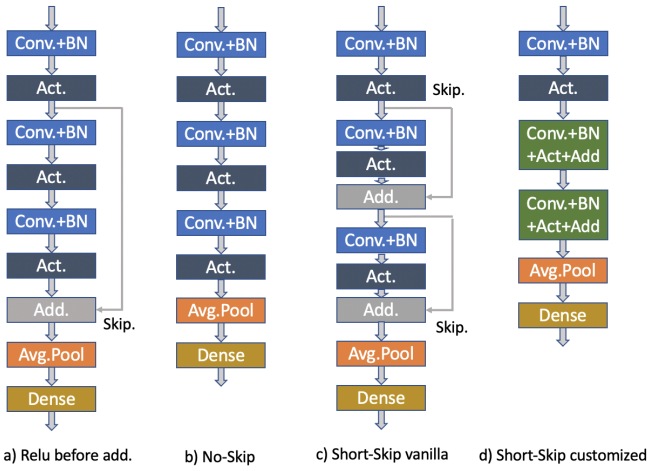


Figure 4.4. layers generated by hls4ml [31] for: a) *Relu before addition* introduced in [40], b) similar model to (a) without the skip connection, c) vanilla *Relu before addition* with short skip connections generated by hls4ml, and d) our customized *Relu before addition* with short skip connections (*Add*), convolution, BN, and activation layers as one new layer for optimizing BRAM utilization. All utilizations are reported at Table 4.5.

for our short skip connections. Long and short skip connections for this configuration are shown in Figures 4.4.a and 4.4.c, respectively.

Using hls4ml to generate these layers results in higher resource utilization for short skip connections (Table 4.5). This is because of the extra *Add* and its buffer. Therefore, inspired by [130], we combine the (Conv.+BN/Act./Add) layers into one layer. The pseudo-code of this new block is provided in Listing 4.3.

The new layers (Figure 4.4.d) reduce the BRAM utilization by 16% compared to long skip connections (Table 4.5) and results in better training results shown in Tables 4.1, 4.2, 4.3, and 4.4.

Listing 4.3. pseudo code for the green blocks in Fig. 4.4.d

```
conv+BN loop:
    acc = apply_filters(input, weights)


output loop:
    res = input //short skip connection
    if(acc>0){res += acc} //Relu
    output = res
```

This new block also can be used for DLAs to reduce the off-chip memory bandwidth.

Figure 4.2 shows the block design of a DLA used in [76] and [75]. It is designed for long skip connections and requires loading the skip connection data from off-chip memory for the element-wise addition. A new DLA with short skip connections can benefit from the green blocks in Figure 4.4.d to eliminate the extra load for the skip connection's data. Therefore, could have a lower utilization on off-chip memory bandwidth.

4.3.3 Quantization

In order to map our models onto an FPGA efficiently, we must quantize the weights to a lower bitwidth and adjust them from floating point to fixed point. In general, FPGA implementations avoid using floating point because it costs significantly more resources than fixed point does. Quantizating deep neural networks with minimal accuracy loss [86, 146] is a tedious and time-consuming task that requires meticulous hyperparameter fine-tuning. To that end, we quantize SKIPTRIM ResNet20 (all skip connections removed) on CIFAR-10 from floating point 32 to two common fixed point bitwidths (<16,4> and <8,3>) via quantization-aware training using QKeras [16].

In Table 4.6, we find that through quantization-aware training our models achieve comparable accuracy when reducing our weight bitwidths from float32 to fixed point <16,4> and <8,3>. With such close quantization accuracy results, the SKIPTRIM method holds under quantization for ResNet20 on CIFAR-10. Ultimately, how much accuracy loss quantization causes is highly dependent on the model and its application.

Table 4.6. Synthesis results for fixed point precision designs on CIFAR10

Design, Quantization	QKeras Acc. (%)	Utilization (%)			
		BRAM	DSP	FF	LUT
ResNet20, <16,4>	91.49	114	12	20	55
ResNet20 (Skip), <16,4>	90.50	96	12	19	47
ResNet20, <8,3>	91.27	55	∼0	28	69
ResNet20 (Skip), <8,3>	89.75	45	∼0	24	60

In Table 4.6, we report the Alveo U250 board resource utilization for <16,4> and <8,3>

quantized ResNet20 models and the accuracy that we achieved in QKeras. Again we see significant drops in BRAM usage. <16,4> SKIPTRIM ResNet20 uses 18% fewer BRAMs, allowing ResNet20 to actually fit onto the board, and <8,3> SKIPTRIM ResNet20 uses 10% fewer BRAMs. The <8,3> model has a slightly lower reduction in BRAMs because when the bitwidth drops below 10 bits, more multiplications on the FPGA are performed on LUTs and FFs, as seen in the <8,3> models' increased LUT and FF usage and decrease DSP usage compared with the <16,4> models [1].

4.4 Limitations and Future Work

We believe that it is important for every work to state its limitations. We carefully performed our experiments on a set of datasets and tested SKIPTRIM on models with different depths. Although we did not observe an example of our method's failure, we cannot guarantee that this method could work for all applications. As mentioned in the text, training quantized networks is a time-consuming task. Therefore, we only provided one example on ResNet20 as a proof of concept.

Looking ahead, we plan to train more quantized models and test larger networks on hardware and provide power measurements for our implemented models for a better exhibit of SKIPTRIM's benefits.

4.5 Conclusion

In this work, we introduced Skiptrim, a method for making ResNets more hardware-efficient for inference on FPGAs. This method can prune all the skip connections from smaller models, and for larger ones, it divides a long skip connection into multiple short ones. We provided optimization for realizing short skip connections on FPGAs and showed that removing or shortening skip connections is worthwhile. On average, SKIPTRIM reduces the BRAM utilization by 20% when all the skip connections are pruned and 16% by shortening them. These

resource reductions are further amplified by the increase in accuracy over ResNets that we find with trimming skip connections on different datasets, such as CIFAR-100, RadioML.2018, and SVHN. The resulting models are indeed topologies that lend themselves to more efficient FPGA implementations.

This chapter, in part is currently being prepared for submission for publication of the mate-rial. Alireza Khodamoradi, Olivia Weng, Nojan Sheybani, Kristof Denolf, Farinaz Koushan-far,and Ryan Kastner. The dissertation author was the primary investigator and author of this material.

Chapter 5

Auto Tuning the Learning Rate

Training a neural network is a time-consuming process that often requires a great deal of optimization of the hyperparameters to achieve a high-quality result. For example, in supervised learning, one has to select an initialization method to start the training, a cost function and an optimizer to perform the training, and a budget of epochs or a target accuracy to stop the training. The training process also involves other choices, such as input normalization, pruning methods, Etc.

Each method or function selected for training comes with a set of hyperparameters that must be tuned. For example, Stochastic Gradient Descent (SGD), which forms the core of many training algorithms, is defined as:

$$\theta_{i+1} = \theta_i - \lambda_i g(\theta_i) \tag{5.1}$$

SGD iteratively updates the network parameters θ (e.g., weights and biases) by multiplying the *learning rate* λ by the derivative of the cost function $g(\theta)$ and subtracting it from the parameters. The training script calculates the cost function using a subset of the training set. The size of this subset is called *batch size*. Throughout this work, we use $\nabla F(\theta)$ to refer to the gradient of the cost function in Batch Gradient Descent and $g(\theta)$ to refer to the gradient of the cost function in SGD.

Unfortunately, there are no concrete rules to select the exact values for hyperparameters.

Moreover, their optimum values[1] heavily depend on the application, the network topology, and choices made for other training parameters. For example, applying quantization to the network parameters requires re-adjusting both hyperparameters (learning rate and batch size) in Equation 5.1 [65].

Ideally, users can fallback on existing hyperparameters that were meticulously tweaked by experts. However, if this fails to achieve the required results, the user resorts to guessing the initial hyperparameters and proceeds to fine-tune the parameters [41, 47]. This tuning process is often a time-consuming task whose outcome depends on the initial guess, user experience, and a bit of luck.

The learning rate is one of the essential hyperparameters in a training process [56, 80, 81]. In this work, we aim to reduce the complexity of tuning the learning rate. We introduce an Adaptive Scheduler for Learning Rate (ASLR) that automatically adjusts the learning rate throughout the training process. Our scheduler is particularly useful for training a network with no provided learning rate since it has only one hyperparameter to tune. Our adaptive learning rate scheduler achieves competitive results compared to existing state-of-the-art and (manually) fine-tuned schedulers with multiple user-defined parameters.

The primary contributions are as follows.

- We introduce a novel adaptive learning rate scheduler with a single user-defined parameter and low tuning complexity. This algorithm can achieve competitive results compared to hand-tuned schedulers and line search methods, and its computational cost is trivial.
- We release our code as open-sourced to enhance accessibility and aid in future comparisons of our work.[2]

The remainder of this chapter is organized as follows. Section 5.1 provides the necessary background material related to the complexity of learning rate tuning. Common trends and

[1] It cannot be proved that hyperparameter values are optimum. Therefore it is loosely used to refer to their acceptable values.

[2] github.com/Xilinx/AdaptiveSchedulers

techniques for learning rate adjustment are reviewed in Section 5.2. Our proposed algorithm is explained in Section 5.3. Experiment results are provided in Section 5.4 and conclusions are provided in Section 5.5.

5.1 Complexity of Learning Rate Tuning

Learning rate is perhaps the most important hyperparameter to tune [35], and in general, it is not possible to calculate the best learning rate a priori [102]. In the following, we provide a brief review of why learning rate tuning is complex and vital.

Complexity of loss surface

Gradient descent algorithm iteratively updates the network parameters by using the first derivative of a cost function. This process provides a direction and a value for changing each trainable parameter to minimize the cost function. However, the first derivative provides a rough estimate of the underlying curvature. To improve this process, one can use the learning rate to control the magnitude of the change. A large learning rate causes more significant changes to the parameters, while a small learning rate results in smaller changes at each step of the training process.

Figure 5.1 shows a simple example of a loss function and its underlying surface (i.e., loss surface). Function F has only one parameter θ, and its underlying curvature has one dimension. At step 1, the red arrow indicates the direction of change for θ to decrease the value of F. The learning rate controls the amount of change in that direction. In this example, if the learning rate is set too small, the search process will get stuck around the local minima (θ_L), and the optimal minima (θ_O) will not be obtained. A large learning rate will result in the search moving too far in the wrong direction (away from the optimal result θ_O), making the search process longer and possibly leaving it to diverge.

The complexity of a loss surface calculated for a network directly correlates with the number of trainable parameters in that network. In a real-world network, the curvature of a loss

surface can depend on tens of millions of parameters [38, 45, 116, 139]. Therefore calculating or estimating a "good" learning rate can be a challenging and expensive-to-compute task [92].

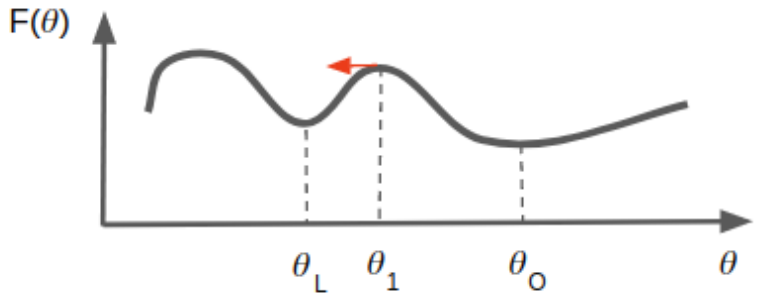


Figure 5.1. A sketch of a loss surface with only one parameter. At $\theta = \theta_1$ the red arrow shows the direction of change in θ for descending F. The learning rate controls the size of the change. A small learning rate holds the optimization process around θ_L. A large learning rate results in a value farther from θ_O.

Dependency on other training parameters

Equation 5.1 is typically augmented in an attempt to improve the training results. A common technique used for enhancing SGD is *momentum*, which regularizes the changes at each step based on the variance of the cost function:

$$\begin{aligned} \theta_{i+1} &= \theta_i - \lambda_i v_i \\ v_i &= \beta v_{i-1} + (1-\beta) g(\theta_i) \end{aligned} \tag{5.2}$$

If used correctly, adding momentum improves the training process. At the same time, it adds additional hyperparameters that must be tuned, e.g., momentum has one additional tunable parameter, β. Other techniques to improve training also introduce new hyperparameters, e.g., weight decay [60]. In many cases, these hyperparameters can affect each other. For example, the learning rate loosely affects the momentum [119] and is strongly related to the batch size [120].

Training a neural network typically requires the tuning of tens of hyperparameters. A

large number of hyperparameters makes the tuning process more challenging. Tuning these hyperparameters is a NP-complete problem [18, 92]. For example, hyperparameters available for training the most popular networks result from many trial and error attempts made by many contributors and are considered "finely-tuned". However, for any of those pairs (set of hyperparameters, network), it cannot be proved that the set is optimal for training its network. It is possible that another set of hyperparameters exists that can improve the training results for the network.

One way to relax this complexity is to reduce the number of hyperparameters. This reduction should not reduce the network's performance. In this work, we introduce a learning rate scheduler with a single user-defined parameter and demonstrate our proposed technique's performance by comparing its results with state-of-the-art techniques with finely-tuned parameters.

5.2 Common Practices for Learning Rate Tuning

In the following, we review four main trends in learning rate tuning, including their complexity, benefits, and disadvantages. Our work is inspired by these methods.

Before starting our review, we should clarify that in each method, only a number of user-defined parameters require careful tuning. These parameters are commonly referred to as *hyperparameters*. Other parameters that require trivial or no tuning are frequently referred to as *default parameters*.

5.2.1 Second Order Information

If the loss function $F(\theta)$ is infinitely differentiable at θ, the result of a small change in its input can be calculated using a Taylor decomposition:

$$F(\theta+\delta\theta)-F(\theta)=\frac{\delta\theta}{1!}F'(\theta)+\frac{(\delta\theta)^2}{2!}F''(\theta)+R_2(\theta) \tag{5.3}$$

Here, $R_2(\theta)$ is the Taylor remainder of order two. A good estimate for learning rate can be calculated by assuming $R_2(\theta) \approx 0$, taking a derivative with regards to $\delta\theta$ from both sides, and setting $\partial(F(\theta+\delta\theta)-F(\theta))/\partial\delta\theta$ to zero:

$$0 = F'(\theta) + \delta\theta F''(\theta) \tag{5.4}$$

Solving Equation 5.4 for $\delta\theta$, results in $-F'(\theta)/F''(\theta)$. Rewriting it in a more familiar form yields: $\delta\theta = -\nabla F(\theta)/H$. By comparing this result with the batch gradient descent equation $\theta_{i+1} = \theta_i - \lambda\nabla F(\theta)$, it can be concluded that an optimum learning rate is equal to the inverse of Hessian matrix of F.

Although that $\lambda = H^{-1}$ can provide a good approximation for learning rate[3], calculating the inverse of a large Hessian matrix is expensive. Moreover, using second order information in training increases the sensitivity to sharp minima. We discuss these drawbacks in more detail in the following.

Complexity

Calculating the inverse of the Hessian matrix has $O(n^3)$ complexity[4] where n is the number of trainable parameters in the network. Calculating this for modern networks with tens of millions of trainable parameters is computationally infeasible. An approximation of the Hessian matrix can be calculated by estimating its largest eigenvalue and the corresponding eigenvector using the power iteration method. However, the cost is still $10\times$ greater than a single calculation for gradient [80]. To improve this approximation, some methods use a layer-wise approximation. While a layer-wise approach can improve the training results, it can also increase the number of user-defined parameters. For example, the method proposed by [138] has 13 user-defined parameters.

The complexity of using second order information for estimating the learning rate can

[3] We assumed $R_2(\theta) \approx 0$.

[4] By using optimized CW-like algorithms, this complexity can be reduced to $O(n^{2.373})$

be relaxed to $O(n^{1.5})$ [81]. Instead of calculating the Hessian matrix, they introduce a notion of distance, $G(\theta)$.

By solving $G(\theta)$ using Kullback–Leibler divergence, they achieved a Fisher matrix for $G(\theta)$ that can be factorized into a Kronecker product of two smaller matrices. Then they create a model based on the approximated distance. By comparing this model with training results, they determine a trust region (a norm ball) to control and adjust the learning rate at each step.

Sensitivity

Techniques that utilize second order information are generally sensitive to sharp minima [24, 54]. Using second order information may also lead to a reduction in the generalization of the network accuracy. It is unclear why deep neural networks generalize well [144], but one common belief is that SGD finds wide minima, which in turn tends to generalize better [43, 54]. Thus, it may be beneficial to avoid such sharp minima, and utilizing second order information makes that less likely.

5.2.2 Adaptive Optimization Methods

A popular approach for regularizing the learning rate throughout the training is extracting useful information from previous steps [27, 56, 74, 101]. This can be done by using averaging methods and estimating the first or second moments (or both) of the gradient. Exponential Moving Average (EMA) is a commonly used technique employed in these methods:

$$EMA_i(g) = \beta EMA_{i-1}(g) + (1-\beta)g(\theta_i) \tag{5.5}$$

This equation calculates a biased average. Dividing the result by $(1-\beta^i)$ can correct the bias.

Using a moving average results in smaller values when the input (g in Equation 5.5) has a large variance. For example, SGD with momentum (SGDM) uses a single *EMA* to dampen the learning rate when the variance is too high (Equation 5.2).

ADAM [56] uses two *EMA*s:

$$\theta_{i+1} = \theta_i - \lambda \frac{\widehat{EMA}_i(g)}{\sqrt{\widehat{EMA}_i(g^2)} + \varepsilon} \tag{5.6}$$

where $\widehat{EMA}$ is bias-corrected *EMA*. Moving averages can calculate an expectation for their input: $\widehat{EMA} \approx E[g]$. And the first and second moments are related as: $E[g^2] = E^2[g] + \text{var}(g)$. Therefore, the fractional portion in Equation 5.6 has an inverse correlation with $\text{var}(g)$ and the effect of the learning rate (λ) is regularized based on the variance of g.

With an increase in user-defined parameters, adaptive optimization methods can provide a fast decay in cost function at the beginning of the training. However, in some cases, they produce a poor generalization [74]. In the following, we review this drawback for these methods.

Reducing variance

There are two sources for variance when calculating SGD ($g(\theta)$). One is due to the underlying pathological curvature of the loss function. The other is related to the sampling of mini-batches that do not fully represent the entire data set. The variance plays an essential role in the optimization process, as we describe in the following.

Referring back to Figure 5.1, the derivative of $F(\theta)$ provides the direction of search. If the learning rate is not small enough, it is unlikely to reach θ_L; the search process will cause θ to move back and forth near θ_L. This increases the variance in $\nabla F(\theta)$. Using an adaptive optimization method, an increase in variance can reduce the effect of the learning rate. E.g., in Equation 5.2, an increase in variance results in a smaller v_i and, therefore, λv_i becomes a smaller value, which forces the algorithm to take smaller steps.

However, for large datasets, $\nabla F(\theta)$ is not used. Instead, its stochastic estimate, $g(\theta)$, is calculated using mini-batches in the SGD algorithm. This estimation itself comes with a variance [120]:

$$\text{SGD fluctuation} \propto \frac{\text{learning rate}}{\text{batch size}} \tag{5.7}$$

In many applications, the SGD variance - more commonly known as the *SGD noise* - can improve the training results. Using a smaller batch size (which typically results in higher SGD noise) is encouraged for achieving a better generalization [82]. For these applications, moving averages can depress the generalization by reducing the SGD noise.

5.2.3 Schedulers

By using a set of user-defined parameters, schedulers adjust a global learning rate or a set of per-layer learning rates (in exchange for an increase in the complexity of hyperparameter tuning) throughout the training process. For example, in multi-step decay, the user sets a starting value, a set of milestones, and a set of decays for each milestone, to adjust a global learning rate during the training process.

A scheduler provides a way to adjust the learning rate at virtually every step of the training. The main disadvantage of using schedulers is their tuning process. Typically, a user starts with a guess or a suggestion from the literature and fine-tunes these parameters using their experience and trial and error. This process can be very time-consuming.

Another common practice for learning rate adjustment is pairing a scheduler with an adaptive optimizer. While this can combine both approaches' benefits, it also requires fine-tuning user-defined parameters for both the scheduler and the optimizer in addition to selecting the right combination for the (scheduler, optimizer) pair.

5.2.4 Methods with Line Search

These methods monitor one of the training metrics, such as validation or training loss, to adjust the learning rate during the training [35]. A variety of line search methods have been proposed in previous work. L4 [105] requires five user-defined parameters. It maintains a minimum attainable loss throughout the training, and by locally linearizing the loss at each step,

it solves a linear equation to calculate the next best learning rate. L4 can be unstable [131].

In [79], authors use a probabilistic belief over the Wolfe conditions [135] to monitor the descent and use a line search to calculate the next best learning rate. This line search requires second order information [5]. As mentioned in Section 5.2.1, using second order information can be costly and sensitive to sharp minima.

A less computationally expensive estimation of an upper bound for a good learning rate can be obtained from the *Armijo condition* [5]. Based on this condition a "good" learning rate should give a *sufficient* decrease in loss function (Figure 5.2)[6]:

$$F(\theta_i + \lambda p_i) \leq F(\theta_i) + c\lambda \nabla F_i^T p_i \tag{5.8}$$

Here, p_i is the direction of change, $0 < c < 1$, and $\nabla F_i^T p_i$ is the directional derivative. Line search algorithms benefit from Armijo condition to search for a good learning rate [92].

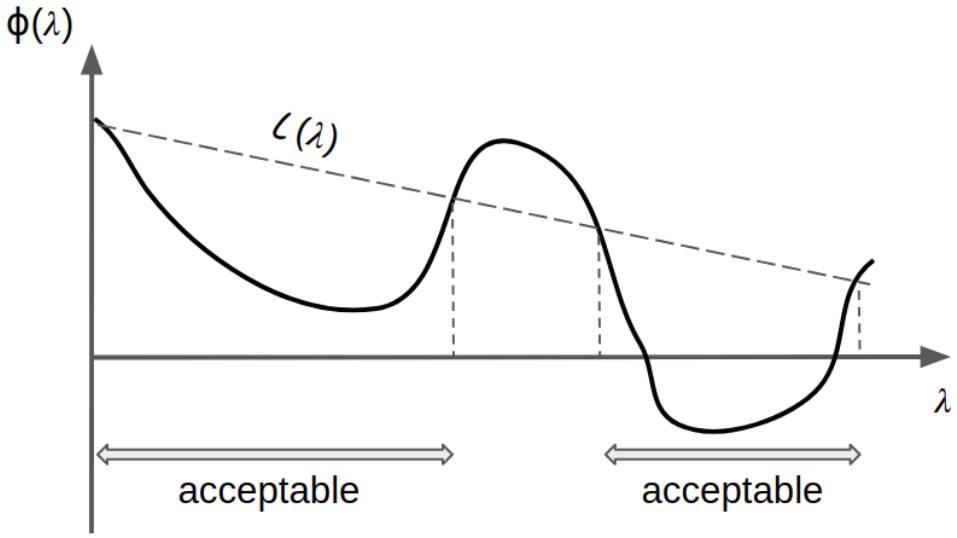


Figure 5.2. Armijo condition: a "good" learning rate should give sufficient decrease in loss function. Here, $\Phi(\lambda) = F(\theta_i + \lambda p_i)$ and $l(\lambda) = F(\theta_i) + c\lambda \nabla F_i^T p_i$ are left and right sides of Equation.5.8, respectively. Acceptable values for "good" λ are when $\Phi(\lambda) < l(\lambda)$.

[5] http://tinyurl.com/probLineSearch

[6] Figure is created based on a drawing from [92]

More straightforward line search methods have shown better results. Authors in [131] use the Armijo condition in a line search and showed improvements on SGD and faster convergence compared to previous work [21, 27, 51, 56, 110, 141]. ASLR relates to this category because of its search algorithm. And it differs from this category because it does not perform a search after each step. Instead, it schedules a learning rate for the next epoch. Therefore, ASLR has a trivial computational cost.

5.3 ASLR

The intuition behind using schedulers is that there exists a global learning rate (or a set of learning rates, one per layer) that can produce a target result for a training process. By fine-tuning the scheduler parameters, a user tries to find these ”good” learning rates based on the training process's outcome and using a validation set.

Adjusting learning rate based on changes in training error is likely to result in poor generalization similar to adaptive optimization methods (Section 5.2.2) and second order information (Section 5.2.1). As mentioned in [35], the learning rate should decay each time the validation error plateaus.

Theoretically, a decaying learning rate is necessary to guarantee convergence of SGD [104]. It is empirically shown that keeping the learning rate constant or decaying it cautiously often works better [79]. Also, a decay-only policy may get stuck around a local minima (Figure 5.1). In ASLR, similar to methods based on the second order information, we allow both increase and decrease in learning rate throughout the training.

In our proposed scheduler, a user fine-tunes a starting value for the learning rate, and then after each epoch, the validation error is monitored. If the validation error plateaus, a simple search algorithm starts to adjust the learning rate. This adjustment continues after every next epoch and stops as soon as an improvement is observed in validation error (see Algorithm 2). We explain each part in more detail in the following.

Estimating the Starting Value

From section 5.2.1, H^{-1} can provide an accurate estimate for per-parameter learning rates λ. Let's assume λ_g (a scalar) is a good global learning rate. And $\lambda_j, (0 \leq j < n)$ are optimal per-parameter learning rates with n being the total number of trainable parameters in the network.

Let's $\lambda_{max} = \max_j\{\lambda_j\}$ be the upper bound and $\lambda_{min} = \min_j\{\lambda_j\}$ be the lower bound for learning rate. A reasonable per-layer learning rate, λ_g, must satisfy the following condition:

$$\lambda_{min} \leq \lambda_g \leq \lambda_{max} \tag{5.9}$$

The right inequality is from Armijo condition and the left inequality is from *curvature condition* [134]. Together, they are referred to as *The Wolfe conditions* [135].

A user can find an initial learning rate that satisfies Equation 5.9 with a simple *learning rate range test* [119]: running the training for a few epochs while increasing the learning rate linearly. By checking the accuracy against the learning rate, one can observe the boundaries for a reasonable starting value. Then the user can select a value between those boundaries, for example, the middle point.

Adjusting Process

The adaptive algorithm starts with the user-specified initial value for the learning rate. When there is no improvement in training results, it searches for the next good learning rate using a simple search algorithm shown in Algorithm 2. Figure 5.3 illustrates an example of learning rate adjustment with ASLR.

In Algorithm 2, an update to the learning rate is only possible after processing one epoch. A search for a better learning rate can potentially be possible after any step of the training. However, since most training scripts use SGD and not Batch Gradient Descent, the results of each step include SGD fluctuation (Equation 5.7). Our proposed algorithm updates the learning rate between epochs to dampen this noise and avoid the evaluation's cost after every step.

Algorithm 2: ASLR Search Algorithm

Require: initial learning rate c
min_cost $\leftarrow 1$
search_direction $\leftarrow 1$
search_range $\leftarrow 1$
search_steps $\leftarrow 0$
while training **do**
process one epoch and for each mini-batch generate new per-layer learning rates (c_u in Equation 5.10)
cost $\leftarrow$ validation cost
if min_cost $<$ cost **then**
$s \leftarrow$ Equation 5.10
if $c +$ search_direction $\times s = 0$ **then**
$c \leftarrow 0.9 \times c$
else
$c \leftarrow c +$ search_direction $\times s$
end if
search_steps $\leftarrow$ search_steps$+1$
if search_steps $=$ search_range **then**
search_range $\leftarrow$ search_range$+1$
search_steps $\leftarrow 0$
search_direction $\leftarrow (-1) \times$ search_direction
end if
else
min_cost $\leftarrow$ cost
search_direction $\leftarrow 1$
search_range $\leftarrow 1$
search_steps $\leftarrow 0$
end if
end while

Step Size in Learning Rate Adjustment

Extremely small changes cannot be applied to the learning rate because each time ASLR adjusts the learning rate, the search process may take several epochs of the training to reach a good learning rate. Therefore we have no choice other than adding discontinuity to the learning rate and apply a feasible change to the learning rate while adjusting it in our algorithm.

Heuristically, we observed that feasible changes in the current learning rate, c, should be equal to $10^{\lfloor \log_{10} c \rfloor}$.

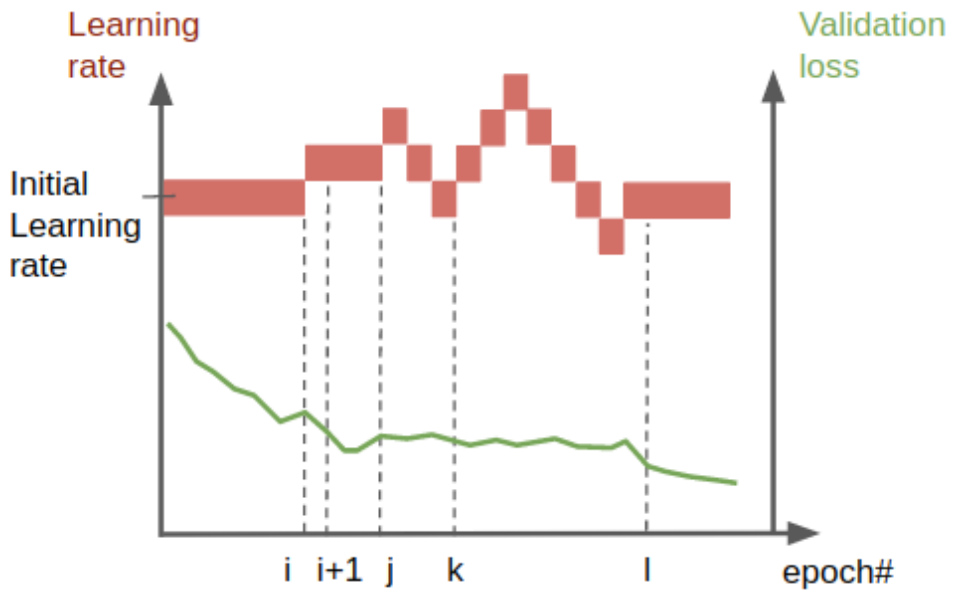


Figure 5.3. Adjusting learning rate. With no improvement in validation loss at epoch i, learning rate increases by s (Equation 5.10). Search stops after the improvement in validation loss at epoch $i+1$. At epoch j, improvement stops again and similar to before, current learning rate increases by s. With no improvement in validation loss at the next epoch, *search_region* increases by one and *search_direction* changes. This happens again at epoch k. And continues until epoch l when there is an improvement in validation loss.

Search Direction and Search Range

As shown in Algorithm 2, each time that the *search_range* is increased, *search_direction* is changed. This mechanism helps to scan a range around the current learning rate for finding a good learning rate. As reported in [74], adaptive methods can suffer from generating extreme values for the learning rate. By gradually increasing the range and changing the direction, we minimize our chance of generating extremely large or extremely small learning rates.

Drawing the Learning Rate from a Uniform Distribution

The starting learning rate (provided by the user) and any other value calculated by the algorithm is, at best, an estimation for a good learning rate. The discontinuity created by the step size applies a limit on these estimated learning rates. For example at $c = 0.05$, the step size is $10^{\lfloor \log_{10} 0.05 \rfloor} = 0.01$. If the *search_direction*=1, the next possible value for the learning rate is

0.06. In ASLR, we will not ignore all possible values between 0.05 and 0.06.

As mentioned in Section 5.2.1, authors in [81] explained how to (more) efficiently use second order information to calculate a trust region (norm ball) and use it to control and adjust the learning rate at each step. Motivated by their work, we fix a range around our estimated learning rates and draw per-batch and per-layer learning rates from that region. The region used in ASLR is a uniform distribution centered at the current learning rate with a width equal to the step size. With c being the current learning rate, for each step of the training, a per-layer learning rate, c_u, is calculated and provided to the optimizer as following:

$$\begin{cases} c_u \sim \mathscr{U}(c-\frac{s}{2}, c+\frac{s}{2}) \\ s = 10^{\lfloor \log_{10} c \rfloor} \end{cases} \tag{5.10}$$

This way, when $c = 0.05$, we draw our learning rates from $\mathscr{U}(0.045, 0.055)$ and when $c = 0.06$, we draw our learning rates from $\mathscr{U}(0.055, 0.065)$ (see Figure 5.4).

Because $\mu(c_u) = c$, our effective learning rate [120] for processing each epoch is still equal to c.

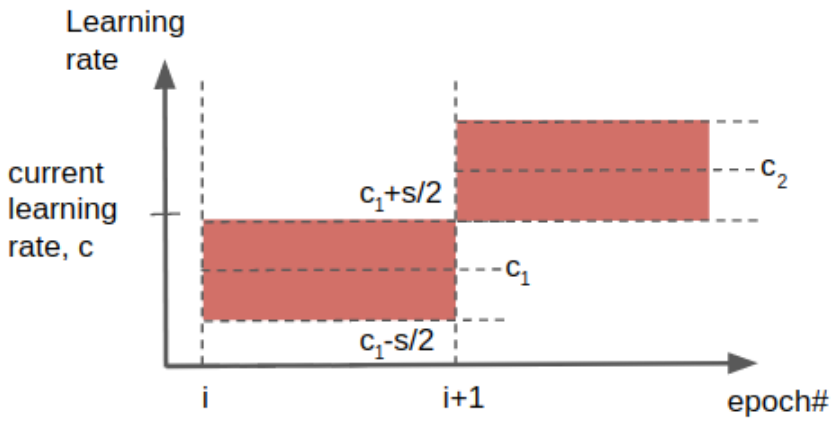


Figure 5.4. Drawing learning rates from a uniform distribution. At each step, per-layer learning rates, c_u are drawn from $\mathscr{U}(c_1 - \frac{s}{2}, c_1 + \frac{s}{2})$. When c_1 is the learning rate for epoch i, calculated with Algorithm 2.

Limitations

We believe that it is essential for every work to state its limitations. We carefully performed extensive experiments and repeated all the reported tests multiple times. Our proposed algorithm is tested on a variety of models ranging from quantized and custom networks to popular networks with Non-quantized parameters, including very deep networks such as the VGG family and residual networks such as the ResNet and DenseNet families. We have also open-sourced our code to allow reproducibility.

Although we did not observe an example of our algorithm's failure, we can not prove or guarantee that this algorithm is superior to all other manually fine-tuned schedulers for all network topologies and training scripts.

We also can not provide a precise comparison between the time spent on tuning our algorithm parameter (starting value) against the time spent on tuning other schedulers' parameters due to the lack of reporting such processes in the literature.

5.4 Results

To evaluate ASLR we used a selection of moderate and hard to classify datasets consisting of ImageNet [23] and both CIFAR10 and CIFAR100 [58] datasets. We selected a variety of different network architectures, including very deep architectures, networks with skip connections, and dense architectures.

We also tested ASLR on networks with quantized parameters. This is a significant test result because hyperparameters of a network must be re-tuned after the quantization is applied [35, 114]. Our results show that ASLR can be employed to train quantized networks with no additional tuning.

Unfortunately, there are no widely recognized benchmarks to use for comparison. Therefore, in our setup, we use publicly available implementations to evaluate ASLR against other work.

In all of our experiments, the reported accuracy results are average over three runs with different seeds. We also set ASLR's initial learning rate similar to the initial rate of the network that we compared against and therefore did not have to perform the initial learning rate search described in Section 5.3.

In the following, we first describe our results for comparing ASLR against line search methods, which includes test results on CIFAR10 and CIFAR100 datasets using ResNet34. We then compare ASLR and different schedulers on CIFAR10 and ImageNet on various network topologies, including quantized networks.

Comparing with Line Search Methods

To compare our work with methods mentioned in Section 5.2.4, we used the implementation [7] that is described at [131] and integrated ASLR into this implementation.

Table 5.1. Comparing validation accuracy of ASLR with Line Search Methods on ResNet34

Dataset	Batch Size	L4	SGD_Armijo	ASLR
CIFAR10	64	87.5%	93.4%	93.6%
	128	86.2%	93.6%	94.2%
CIFAR100	64	63.7%	73.8%	74.5%
	128	60.8%	74.8%	75.7%

Table 5.1 shows a comparison between our results and two other line search methods. To generate these results, we set ASLR's initial learning rate to 0.1 (with no additional tuning) and total epochs to 150. Accuracy results are average over three runs with different seeds.

An interesting observation is the processing time between the three methods. At each step, ASLR draws the learning rates from a uniform distribution. Whereas L4 [105] and SGD_Amijo [131] have to do a line search. Compared to ASLR, these line search methods required extra time for processing each epoch. We calculated the average of per-epoch additional

[7] https://github.com/IssamLaradji/sls

time needed for these methods on a desktop machine with one GPU for all the training epochs. Table 5.2 shows our results.

Table 5.2. Comparing average training time per epoch between ASLR, L4, and SGD_Amijo (CIFAR10 and ResNet34).

	Training time per epoch (Seconds)		
batch size	ASLR	L4	SGD_Amijo
64	85	119	129
128	84	113	127

The validation accuracy evolution curve for experiments in Table 5.1 is shown in Figure 5.5. The oscillation in ASLR's curve is due to its search algorithm. Each time the validation loss plateaus, ASLR starts its search, and the search range expands after each epoch until an improvement is observed in validation loss. Changes in the learning rate during this search cause the oscillation in its validation accuracy curve.

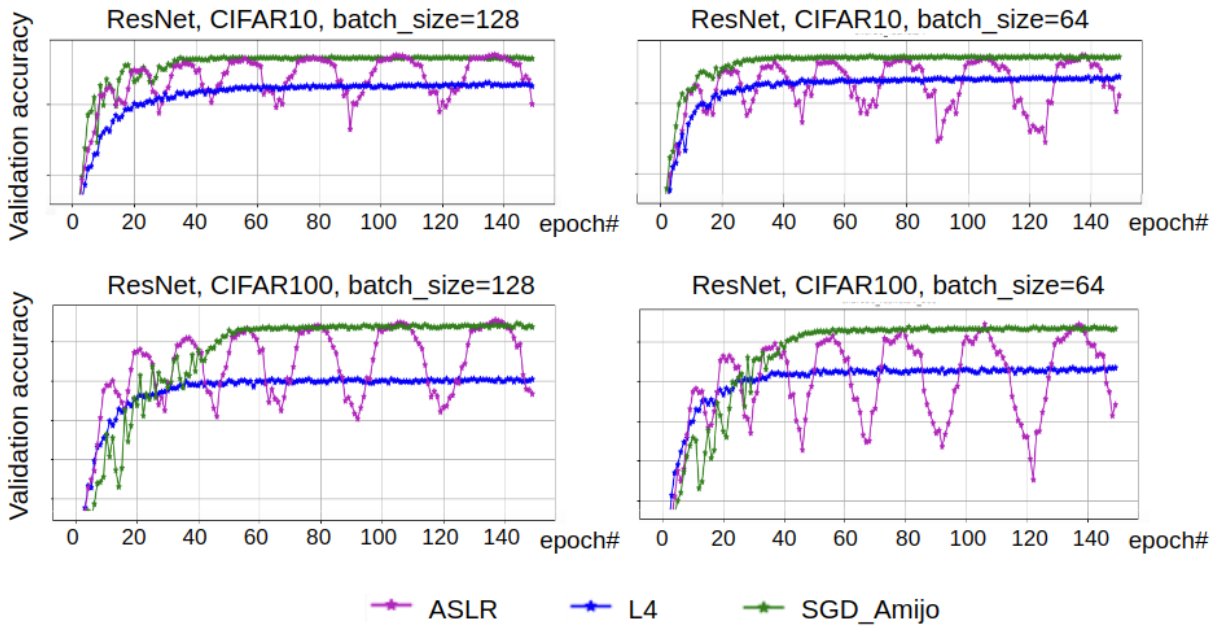


Figure 5.5. Comparison between the validation accuracy evolution curve for ASLR and line search methods: L4 and SGD_Amijo.

Comparing with Schedulers

To our knowledge, there are no widely recognized benchmarks to use for comparing our method with methods described in Section 5.2.3. Therefore for this part of our experiments, we have selected a diverse range of networks with publicly available implementations and already-tuned hyperparameters for CIFAR10 and ImageNet datasets.

CIFAR10: We compared ASLR with state-of-the-art results for a number of networks selected from ResNet [38], DensNet [45], WRN [139], and VGG [116] families. We also selected two reduced precision networks: WRN_1bit and CNV_1bit [130] and a custom VGG11 network [8] where parameters are quantized and fully connected layers are removed to avoid over-fitting for the CIFAR10 dataset.

Table 5.3. Comparing validation accuracy of ASLR with schedulers on CIFAR10.

Network	Scheduler accuracy	ASLR accuracy
Resnet20	92.2%	92.2%
Resnet56	93.3%	93.9%
DenseNet40	92.8%	92.9%
WRN20_1bit	95.2%	94.9%
VGG11_8bits	91.5%	91.4%
VGG11_6bits	91.2%	91.2%
CNV_1bit	78.5%	78.5%

The results of our comparisons are shown in Table 5.3. In the following, we describe the schedulers used to generate the results in *Scheduler accuracy* column.

ResNet20, ResNet56, and DenseNet40 used multi-step-decay scheduler with nine, nine, and five user-defined parameters respectively [9]. WRN20_1Bit used a cosine annealing scheduler with two user-defined parameters [10]. VGG11 used a multi-step decay scheduler with seven user-defined parameters. And CNV_1Bit used a multi-step-decay scheduler with nine user-defined

[8] https://github.com/Xilinx/brevitas

[9] https://keras.io/examples

[10] https://github.com/osmr/imgclsmob

parameters.

To generate the results in *ASLR accuracy* column, we set the initial learning rate of ASLR equal to the initial learning rate of the scheduler that we compared against (the scheduler in the same row of the table). Results are an average of three runs with different seeds. Table 5.3 shows that ASLR can achieve similar or better results compared to highly tuned manual schedulers while having only one user-defined parameter.

Figure 5.6 illustrates a comparison between ASLR and the multi-step-decay scheduler used with ResNet20. Both schedulers achieved similar validation accuracy results. As shown in Figure 5.6, throughout the training, ASLR starts its search earlier than the first decay in the other scheduler, and by the end of the training, it almost follows the finely tuned multi-step-decay. Similar behavior was observed when training other networks in Table 5.3.

ImageNet: We selected three networks from ResNet and VGG families to test ASLR on the ImageNet dataset. ResNet10, ResNet50, and VGG11. Table 5.4 shows the results of our experiments on this dataset. The schedulers used to generate the results in *Scheduler accuracy* column are cosine annealing schedulers with two user-defined parameters [11]. ASLR's initial learning rate was set to the scheduler's initial learning rate in the same row in Table 5.4. All results are an average of three runs with different seeds.

Table 5.4. Comparing validation accuracy of ASLR with schedulers on ImageNet.

Network	Scheduler accuracy	ASLR accuracy
Resnet10	65.5%	66.0%
Resnet50	75.2%	75.2%
VGG11	67.7%	70.9%

[11] https://github.com/osmr/imgclsmob

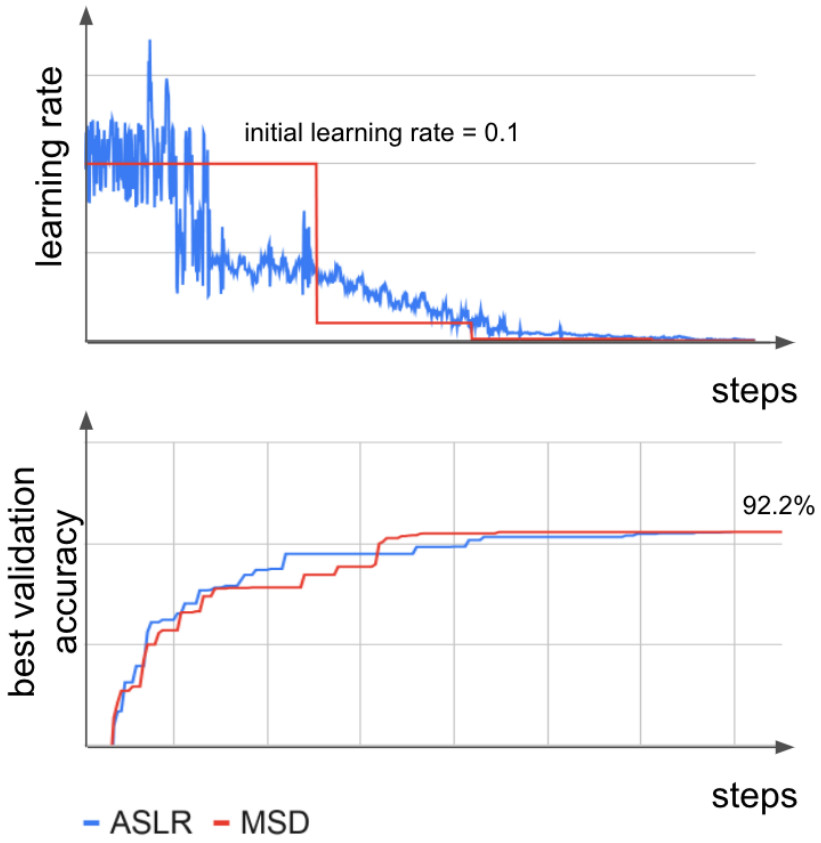


Figure 5.6. Comparison between ASLR and multi-step-decay on ResNet20

5.5 Conclusions

This work provided a brief review of commonly used learning rate adjustment methods and explained their gains and disadvantages. We described the complexity of finding reasonable learning rates and introduced an Adaptive Scheduler for Learning Rate (ASLR) with a single user-defined parameter. We explained how our algorithm adjusts the learning rate during the training process and showed that even though it has a simple algorithm, it can achieve competitive results compared to training scripts with finely-tuned hyperparameters. Our result section provided performance results for ASLR on various network topologies, including custom networks with quantized parameters. The ability to train uncommon and quantized networks is an essential feature of ASLR and shows that this scheduler can train a wide range of network designs. This

feature can reduce the time for testing and designing custom networks by reducing the tuning time spent on hyperparameters for the learning rate. We also showed that ASLR has a smaller computation complexity compared to line search methods.

This chapter, in full, is a reprint of the material as it appears in the International JointConference on Neural Networks 2021. Alireza Khodamoradi, Kristof Denolf, Kees Vissers, andRyan Kastner. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] Thea Aarrestad, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Jennifer Ngadiuba, Christoffer Petersson, Hampus Linander, Yutaro Iiyama, Giuseppe Di Guglielmo, Javier Duarte, et al. Fast convolutional neural networks on fpgas with hls4ml. *arXiv preprint arXiv:2101.05108*, 2021.

[2] Gina Adam, Ali Khiat, and Themis Prodromakis. Challenges hindering memristive neuromorphic hardware from going mainstream. In *Nature Communications*, volume 9, 2018.

[3] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. 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*, 34(10):1537–1557, 2015.

[4] apache. mxnet. mxnet.apache.org. Accessed: 2021-28-03.

[5] L. Armijo. Minimization of functions having lipschitz continuous first partial derivatives. In *Pacific Journal of Mathematics*, volume 16, 1966.

[6] C. Baskin, N. Liss, E. Zheltonozhskii, A. M. Bronstein, and A. Mendelson. Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 162–169, Los Alamitos, CA, USA, may 2018. IEEE Computer Society.

[7] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[8] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[9] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[10] C. Brandli, R. Berner, M. Yang, S. Liu, and T. Delbruck. A 240 x 180 130 db 3u sec latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, 2014.

[11] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend, and spell. *IEEE International Conference on Acoustic, Speech, and Signal Processing (ICASSP)*, 2015.

[12] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[13] Kit Cheung, Simon R Schultz, and Wayne Luk. A large-scale spiking neural network accelerator for fpga systems. In *International Conference on Artificial Neural Networks*, pages 113–120. Springer, 2012.

[14] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[15] CNET. Samsung turns ibm's brain-like chip into a digital eye.

[16] CN Coelho, A Kuusela, S Li, H Zhuang, T Aarrestad, V Loncar, J Ngadiuba, M Pierini, AA Pol, and S Summers. Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders. *arXiv preprint arXiv:2006.10159*, 2006.

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

[18] B. DasGupta and H. T. Siegelmann. On the complexity of training neural networks with continuous activation functions. In *7th ACM Conference on Learning Theory, 1994*, 1994.

[19] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[20] S. Decker, D. McGrath, K. Brehmer, and C. Sodini. A 256 x 256 cmos imaging array with wide dynamic range pixels and column-parallel digital output. *IEEE J. Solid-State Circuits*, 33:2081–2091, 1998.

[21] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, 2014.

[22] T. Delbruck. Scientific: Particle image velocimetry.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[24] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio. Sharp minima can generalize for deep nets. In *arXiv:1703.04933*, 2017.

[25] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. Fast inference of deep neural networks in fpgas for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[26] Javier Duarte, Philip Harris, Scott Hauck, and et al. Fpga-accelerated machine learning inference as a service for particle physics computing. In *Computing and Software for Big Science*, volume 3, 2019.

[27] John Duchi, Elad Hazan, and Yorman Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Journal of Machine Learning Research (JMLR)*, 2011.

[28] Allen Edward. *Kolmogorov-Smirnov Test for Discrete Distributions*. Defense Technical Information Center, Monterey, California, 1976.

[29] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 113(41):11441–11446, 2016.

[30] Sawada et al. Truenorth ecosystem for brain-inspired computing: Scalable systems, software, and applications. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 130–141, 2016.

[31] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarestaad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices, 2021.

[32] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu. Encoding, model, and architecture: Systematic optimization for spiking neural network in fpgas. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[33] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Third generation neural networks: Spiking neural networks. In *Advances in Computational Intelligence*, pages 167–178, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[36] Alex Graves and Jaitly Navdeep. Towards end-to-end speech recognition with recurrent neural networks. In *2014 International Conference on Machine Learning (ICML)*, volume 14, 2014.

[37] K. He and J. Sun. Convolutional neural networks at constrained time cost. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5353–5360, 2015.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv:1502.01852*, 2015.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 630–645, Cham, 2016. Springer International Publishing.

[41] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li. Bag of tricks for image classification with convlutional neural networks. In *arXiv:1812.01187*, 2018.

[42] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]*, March 2015. arXiv: 1503.02531.

[43] S. Hochreiter and J. Schmidhuber. Flat minima. In *Neural Computation*, volume 9, pages 1–42, 1997.

[44] Alan Hodgkin and Andrew Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. In *The Journal of physiology*, volume 117, page 500, 1952.

[45] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *arXiv:1608.06993*, 2018.

[46] Zan Huang, Hsinchun Chen, Chia jung Hsu, Wun hwa Chen, and Soushan Wu. Credit rating analysis with support vector machines and neural networks: A market comparative study, 2004.

[47] F. Hutter, J. Lücke, and L. Schmidt-Thieme. Beyond manual tuning of hyperparameters. In *Künstl Intell 29*, pages 329–337. Springer, 2015.

[48] inilabs. inilabs. https://inilabs.com. [Online; accessed 28-December-2017].

[49] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[50] Eugene. Izhikevich. Simple model of spiking neurons. In *Transactions on Neural Networks, IEEE*, volume 14, pages 1569 – 1572, 2003.

[51] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, pages 315–323, 2013.

[52] X. Ju, B. Fang, R. Yan, X. Xu, and H. Tang. An fpga implementation of deep spiking neural networks for low-power and fast classification. *Neural Computation*, 32(1):182–204, 2020.

[53] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic plasticity dynamics for deep continuous local learning (decolle). *Frontiers in Neuroscience*, 14:424, 2020.

[54] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P.T.P Tang. On large-batch training for deep learning generalization gap and sharp minima. In *arXiv:1609.04836*, 2017.

[55] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[56] D. P. Kingma and J. Lei Ba. Adam: a method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[57] E. J. Knoblock and H. R. Bahrami. Investigation of spiking neural networks for modulation recognition using spike-timing-dependent plasticity. In *2019 IEEE Cognitive Communications for Aerospace Applications Workshop (CCAAW)*, pages 1–5, 2019.

[58] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Toronto University, 2009.

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[60] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *Proceedings of the 4th International Conference on Neural Information Processing Systems*, 1991.

[61] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[62] Yann A. LeCunLéon, Bottou Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop in neural networks. *Tricks of the Trade*, pages 9–48, 2012.

[63] Chang W. Lee and Jung-A Park. Assessment of hiv/aids-related health performance using an artificial neural network. *Information & Management*, 38(4):231 – 238, 2001.

[64] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

[65] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein. Training quantized nets: a deeper understanding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5813–5823, 2017.

[66] X. Li, L. Ding, L. Wang, and F. Cao. Fpga accelerates deep residual learning for image recognition. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 837–840, 2017.

[67] P. Lichesteiner, C. Posch, and T. Delbruck. A 128 x 128 120 db 15u sec latency asynchronous temporal contrast vision sensro. *IEEE Journal of Solid-State Circuits*, 43(2):566–576, 2008.

[68] X. Lin, S. Yin, F. Tu, L. Liu, X. Li, and S. Wei. Lcp: a layer clusters paralleling mapping method for accelerating inception and residual networks on fpga. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[69] A. Linares-Barranco, F. Gómez-Rodriguez, V. Villanueva, L. Longinotti, and T Delbruck. A usb3.0 fpga event-based filtering and tracking framework for dynamic vision sensors. *IEEE International Symposium on Circuits and Systems*, pages 2417–2420, 2015.

[70] Beiye Liu, Yiran Chen, Btyant Wysocki, and Tingwen Huang. Reconfigurable neuromorphic computing system with memristor-based synapse design. *Neural Processing Letters*, 41:159–167, 2015.

[71] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Qing Wu, and Hao Jiang. A spiking neuromorphic design with resistive crossbar. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[72] H. Liu, C. Brandli, S Liu, and T. Delbruck. Design of a spatiotemporal correlation filter for event-based sensor. *IEEE International Symposium on Circuits and Systems*, pages 722–725, 2015.

[73] X. Liu, D. Yang, and A. E. Gamal. Deep neural network architectures for modulation classification. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 915–919, 2017.

[74] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. Adaptive gradient methods with dynamic bound of learning rate. In *International Conference on Learning Representations (ICLR)*, 2019.

[75] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. Optimizing the convolution operation to accelerate deep neural networks on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1354–1367, 2018.

[76] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. Seo. End-to-end scalable fpga accelerator for deep residual networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[77] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. In *1997 Neural Networks*, volume 10, pages 1659–1671, 1997.

[78] Misha Mahowald. *An analog VLSI system for sterescopic vision*. Kluwer, Boston, MA, 1994.

[79] M. Mahsereci and P. Hennig. Probabilistic line searches for stochastic optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, pages 181–189. Curran Associates, Inc., 2015.

[80] J. Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, pages 735–742, 2010.

[81] J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximation curvature. In *arXiv:1503.05671*, 2016.

[82] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. In *arXiv:1804.07612*, 2018.

[83] G. J. Mendis, J. Wei, and A. Madanayake. Deep learning-based automated modulation classification for cognitive radio. In *2016 IEEE International Conference on Communication Systems (ICCS)*, pages 1–6, 2016.

[84] S. Mendis, S. Kemeny, R. Gee, B. Pain, C. Staller, Q. Kim, and E. Fossum. Cmos active pixel image sensors for highly integrated imaging systems. *IEEE J. Solid State Circuits*, 32:187–197, 1997.

[85] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *AAAI*, 2020.

[86] Bert Moons, Koen Goetschalckx, Nick Van Berckelaer, and Marian Verhelst. Minimum energy quantized neural networks, 2017.

[87] Simon W Moore, Paul J Fox, Steven JT Marsh, A Theodore Markettos, and Alan Mujumdar. Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 133–140. IEEE, 2012.

[88] Surya Narayanan, Ali Shafiee, and Rajeev Balasubramonian. INXS: bridging the throughput and energy gap for spiking neural networks. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*, pages 2451–2459. IEEE, 2017.

[89] Surya Narayanan, Karl Taht, Rajeev Balasubramonian, Edouard Giacomin, and Pierre-Emmanuel Gaillardon. Spinalflow: An architecture and dataflow tailored for spiking neural networks. In *2020 47th International Symposium on Computer Architecture (ISCA-47)*, 2020.

[90] Daniel Neil and Shih-Chii Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12): 2621–2628, 2014.

[91] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[92] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer, 2nd edition, 2000.

[93] Thomas Brox Olaf Ronneberger, Philipp Fischer. U-net: Convolutional networks for biomedical image segmentation. *arXiv:1505.04597 [cs]*, 2015. arXiv: 1505.04597.

[94] Timothy J. O'Shea, Johnathan Corgan, and T. Charles Clancy. Convolutional radio modulation recognition networks. In *Engineering Applications of Neural Networks*, pages 213–226, Cham, 2016. Springer International Publishing.

[95] T. J. O'Shea, T. Roy, and T. C. Clancy. Over-the-air deep learning based radio signal classification. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):168–179, 2018.

[96] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, 2013.

[97] Priyadarshini Panda and Kaushik Roy. Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition. In *International Joint Conference on Neural Networks (IJCNN)*, pages 299 – 306, 2016.

[98] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[99] A. Pooresmaelli, G. Cicchini, M. Morrone, and S. Burr. Spatiotemporal filtering and motion illusion. *Journal of Vision*, 13(21), 2013.

[100] J. Pratt and J. Gibbons. *Concepts of Nonparametric Theory*, chapter Kolmogorov–Smirnov Two-Sample Tests. Springer, 1981.

[101] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2015.

[102] R. D. Reed and R. J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, Cambridge, MA, 1998.

[103] A. Rios-Navarro, E. Cerezuela-Escudero, Dominguez-Morales M., A. Jimenez-Fernandez, G. Jimenez-Moreno, and A. Linares-Barranco. Live demonstration: Real-time motor rotation frequency detection by spike-based visual and auditory aer sensory integration for fpga. *IEEE International Symposium on Circuits and Systems*, 2015.

[104] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[105] M. Rolinek and G. Martius. L4: practical loss-based stepsize adaption for deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2018.

[106] Monti R.P., Tootoonian S., and Cao R. Avoiding degradation in deep feed-forward networks by phasing out skip-connections. *Artificial Neural Networks and Machine Learning (ICANN)*, 11141, 2018.

[107] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[108] Ali Samadzadeh, Fatemeh Sadat Tabatabaei Far, Ali Javadi, Ahmad Nickabadi, and Morteza Haghir Chehreghani. Convolutional spiking neural networks for spatio-temporal feature extraction, 2020.

[109] Bharat Bhusan Sau and Vineeth N. Balasubramanian. Deep model compression: Distilling knowledge from noisy teachers. *CoRR*, abs/1610.09650, 2016.

[110] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. In *arXiv:1309.2388*, 2016.

[111] Oleg Semery. Computer vision models on pytorch. pypi.org/project/pytorchcv/. Accessed: 2021-28-03.

[112] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience*, 13:95, 2019.

[113] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[114] Sungho Shin, Yoonho Boo, and Wonyong Sung. Knowledge distillation for optimization of quantized deep neural networks. In *arXiv:1909.01688*, 2019.

[115] Sumit Bam Shrestha and Garrick Orchard. SLAYER: Spike layer error reassignment in time. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1419–1428. Curran Associates, Inc., 2018.

[116] K. Simonyan and A. Zisserman. Very deep convlutional networks for large-scale image classification. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[117] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[118] James Smith. A roadmap for reverse-architecting the brain's neocortex.

[119] L. N. Smith. Cyclical learning rates for training neural networks. In *arXiv:1506.01186*, 2017.

[120] S. Smith, P. Kindermans, C. Ying, and Q. V. Le. Don't decay the learning rate, increase the batch size. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[121] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv:1507.06228*, 2015.

[122] Dmitri Strukov, Gregory Snider, Duncan Stewart, and Stanley Williams. The missing memristor found. In *Nature*, volume 453, pages 80–83, 2008.

[123] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[124] T. Tang, L. Xia, B. Li, R. Luo, Y. Chen, Y. Wang, and H. Yang. Spiking neural network with rram: Can we use it for real-world application? In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 860–865, 2015.

[125] Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1195–1204, Red Hook, NY, USA, 2017. Curran Associates Inc.

[126] David Thomas and Wayne Luk. Fpga accelerated simulation of biologically plausible spiking neural networks. In *2009 17th IEEE symposium on field programmable custom computing machines*, pages 45–52. IEEE, 2009.

[127] H. Tian, B. Fowler, and A. Gamal. Analysis of temporal noise in cmos photodiode active pixel sensor. *IEEE Journal of Solid-State Circuits*, 36(1):92–101, 2001.

[128] Hui Tian. *Noise Analysis in CMOS Image Sensors*. PhD thesis, Stanford University, 2000.

[129] S. Tridgell, D. Boland, P. H. W. Leong, R. Kastner, A. Khodamoradi, and Siddhartha. Real-time automatic modulation classification using rfsoc. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 82–89, 2020.

[130] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, Monterey California USA, February 2017. ACM.

[131] Sharan Vaswani, Aaron Mishkin, Issam Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. Painless stochastic gradient: Interpolation, line-search, and convergence rates. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2019.

[132] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[133] Thomas Voegtlin. Temporal coding using the response properties of spiking neurons. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, page 1457–1464, Cambridge, MA, USA, 2006. MIT Press.

[134] P. Wolfe. Convergence conditions for ascent methods. In *SIAM Review*, volume 11, pages 226–235, 1969.

[135] P. Wolfe. Convergence conditions for ascent methods. ii: some corrections. In *SIAM Review*, volume 13, pages 185–188, 1971.

[136] O. Yadid-Pecht, B. Mansoorian, E. Fossum, and B. Pain. Optimization of noise and responsivity in cmos active pixel sensors for detection of ultra low light levels. *Proc. SPIE*, 3019:125–136, 1997.

[137] F. Yang, Y. Lu, L. Sbaiz, and M. Vetterli. Bits from photons: Oversampled image acquisition using binary poisson statistics. *IEEE Transactions on Image Processing*, 21(4):1421–1436, 2011.

[138] Z. Yao, A. Gholami, D. Arfeen, R. Liaw, J. Gonzalez, K. Keutzer, and M. W. Mahoney. Large batch size training of neural networks with adversarial training and second-order information. In *arXiv:1810.01021*, 2020.

[139] S. Zagoruyko and N. Komodakis. Wide residual networks. In *arXiv:1605.07146*, 2017.

[140] Sergey Zagoruyko and Nikos Komodakis. Diracnets: Training very deep neural networks without skip-connections. *CoRR*, abs/1706.00388, 2017.

[141] Matthew Zeiler. Adadelta:an adaptive learning rate method. In *arXiv:1212.5701*, 2012.

[142] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.

[143] Friedemann Zenke and Surya Ganguli. Superspike: Supervised learning in multilayer spiking neural networks. *Neural Computation*, pages 1514–1541, 2018.

[144] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *arXiv:1611.03530*, 2016.

[145] Y. Zhao, X. Zhang, X. Fang, L. Li, X. Li, Z. Guo, and X. Liu. A deep residual networks accelerator on fpga. In *2019 Eleventh International Conference on Advanced Computational Intelligence (ICACI)*, pages 13–17, 2019.

[146] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients, 2018.