# UC Santa Barbara

## UC Santa Barbara Electronic Theses and Dissertations

### Title

Designing and Enabling Temporal Architectures for Neural Networks

### Permalink

https://escholarship.org/uc/item/1556143k

### ISBN

9798241679819

### Author

Gretsch, Rhys

### Publication Date

2026-02-23

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California

Santa Barbara

# Designing and Enabling Temporal Architectures for Neural Networks

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Rhys Gretsch

Committee in charge:

Professor Timothy Sherwood, Chair

Professor Jonathan Balkind

Professor Kerem Camsari

Professor Michael Beyeler

March 2026

The Dissertation of Rhys Gretsch is approved.

---

Professor Jonathan Balkind

---

Professor Kerem Camsari

---

Professor Michael Beyeler

---

Professor Timothy Sherwood, Committee Chair

February 2026

Designing and Enabling Temporal Architectures for Neural Networks

Copyright © 2026

by

Rhys Gretsch

# Acknowledgements

I would like to express my sincere gratitude to the many people who have helped support my journey through this PhD, as well as those who helped provide a sense of community and belonging in Santa Barbara. I would first like to thank my advisor, Tim Sherwood. None of the contributions in this thesis would be possible without his incredible technical guidance and assistance. Beyond being a brilliant scientist, Tim is an incredibly supportive and kind advisor with a seemingly limitless patience. I would also like to thank the other PI of Archlab, Jonathan Balkind for his support throughout my PhD process. Jon has helped develop an incredibly welcoming and friendly lab culture, and has provided invaluable advice in navigating PhDs, academia, and the computer architecture community. The entire Archlab and all of its members, past and present, have been incredibly helpful, providing incredible collaborators and a constant stream of people willing to grab lunch when I'm too lazy to bring something from home. Special thanks to Alvin, Deeksha, and Chris for helping me as I began my PhD to understand what is required for a successful PhD. Thanks to Pranjali, Naz, Guillem, Parker, Yeganeh, and Naomi for all of their lunch time discussions that broke up the monotony of sitting at a desk and coding or writing.

I would also like to thank my family for always supporting me. Thanks to my Mom and Dad who have always believed in me and initially started me down the path of engineering. My siblings, Daniel, Jessica and Rane, who have all shaped my life in their own way and are my closest friends. All of my aunts and uncles who always provide support and delicious food whenever I'm around. Nicole, thank you for being the most supportive, nice, and caring. Thanks to all my friends I've made in Santa Barbara, and all of the Frisbee teams that have allowed me to play for them. Special thanks to Black Tide, which has allowed me to be involved in the program long beyond my playing days.

# Curriculum Vitæ

Rhys Gretsch

## Education

| | |
|---|---|
| 2026 | Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara. |
| 2023 | M.S. in Electrical and Computer Engineering, University of California, Santa Barbara. |
| 2021 | B.S in Computer Engineering, North Carolina State University |

## Awards

| | |
|---|---|
| 2025 | **IEE Excellence in Research Graduate Fellowship**, UCSB Institute for Energy Efficiency |
| 2025 | **MICRO Top Picks**, IEEE |

## Publications

**Rhys Gretsch**, Michael Beyeler, Jeremy Lau, Timothy Sherwood, "Synthesis and optimization of superconducting circuits with mature toolchains," *International Symposium on Computer Architecture (ISCA)*. 2025.

**Rhys Gretsch**, Timothy Sherwood, "Superconnectors: A Latency Insensitive Approach to SFQ Design," *International Conference on Rebooting Computing (ICRC)*. 2024.

**Rhys Gretsch**, Peiyang Song, Advait Madhavan, Jeremy Lau, Timothy Sherwood, "Energy Efficient Convolutions with Temporal Arithmetic," *International Conference on Rebooting Computing (ICRC)*. 2024.

## Teaching

| | |
|---|---|
| Summer 2024 | Instructor of Record, **INT93-LS**: Introduction to Research-Neuromorphic Computing |
| Spring 2024 | Teaching Assistant, **CS 64**: Computer Organization and Logic Design |
| Fall 2021 | Teaching Assistant, **ECE153a**: Design and Coding of Embedded Programs |

## Abstract

Designing and Enabling Temporal Architectures for Neural Networks

by

Rhys Gretsch

The time in which an event occurs can carry meaningful information when considering its relation to other events. Leveraging this information for direct computation provides an alternative to the current digital paradigm, allowing for a simpler interaction with the physical world. Relationships between the temporal response created by any physical process can be directly computed upon using the same hardware substrate that supports digital logic without expensive conversions to binary representations. This offers the potential for energy efficient computation, but requires appropriate applications and careful hardware organization.

This dissertation presents techniques that allow data to remain in the time domain while performing neural network operations. A general framework of temporal arithmetic is enabled through a negative log transformation with delay-based approximations. Fully temporal large scale, programmable architectures can leverage this framework through the use of hardware recurrence and memory devices that capture the temporal relationship between two signals. Neural network inference can be fully supported by these architectures, and a detailed analysis of the energy-accuracy tradeoff introduced by the architectural decisions is presented. Finally this dissertation explores Zeroth-Order optimization, a technique that can be used to improve the temporal neural networks, and presents a fully digital architecture for energy efficient transformer finetuning.

# Contents

**Curriculum Vitae** **v**

**Abstract** **vi**

**1 Introduction** **1**
1.1 Permissions and Attributions . . . . . 4

**2 Energy Efficient Convolutions With Temporal Arithmetic** **5**
2.1 Introduction . . . . . 5
2.2 Delay Space Arithmetic . . . . . 9
2.3 Recurrence Architecture . . . . . 19
2.4 Rolling Shutter Convolution Architecture . . . . . 22
2.5 Evaluation and Results . . . . . 29
2.6 Conclusion . . . . . 40

**3 Single Spike Artificial Neural Networks** **42**
3.1 Introduction . . . . . 42
3.2 Neural Networks with Delay Space . . . . . 45
3.3 A Temporal/Digital Systolic Architecture . . . . . 53
3.4 Dataflow for Temporal Operation . . . . . 63
3.5 Evaluation Framework . . . . . 65
3.6 Results . . . . . 67
3.7 Conclusion . . . . . 79

**4 Architecture for Energy Efficient ZO Training** **81**
4.1 Introduction . . . . . 81
4.2 ZO Training and Parallelization Opportunities . . . . . 86
4.3 Parallel Processing Elements . . . . . 90
4.4 Difference Based Quantization . . . . . 93
4.5 Architectural Support For Update Reduction . . . . . 97
4.6 Experimental Setup . . . . . 100
4.7 Evaluation . . . . . 102

4.8 Enhancing Temporal Computing . . . . . 114
4.9 Conclusion . . . . . 115

**5 Conclusions and Outlook** **117**

**Bibliography** **120**

# Chapter 1

# Introduction

Time is an inherent property in all physical actions, much to the chagrin of digital designers. Transistors, the building blocks of modern digital logic, take time to switch and propagate the result of the logical function. The standard practice for semiconductor design is to attempt to limit this propagation time as much as possible, going so far as to pretend it doesn't exist in many logic simulations. This is useful for reasoning about the correctness of digital logic, a spatial paradigm that represents each additional bit with physically separate wires, but largely ignores time as a potential dimension in which we can encode information.

Thankfully, exploiting temporal events for information encoding and computation is fairly straightforward using standard CMOS logic. Temporal representation works by controlling when a wire transitions from a low voltage to a high voltage, using either artificial delay elements or integrating directly into sensors that detect when an event occurs. Once a temporal representation has been achieved then logic operators can be repurposed to extract temporal representations. This framework has been shown to be incredibly energy efficient when paired with the right application, such as tropical algebra [1], dynamic programming problems [2], or decision trees [3].

This energy efficiency makes temporal computing an attractive alternative to classical computing as the IT sector's energy consumption is growing at an unsustainable pace. This insatiable demand for power is being driven by the current explosion in machine learning, which has achieved impressive performance and utility across a wide range of tasks. Neural networks with billions of parameters have enabled this new utility, but require trillions of arithmetic and memory operations for a single inference. Training these models requires even more compute and data, requiring data centers to be built in order to service the current demand for both inference and training.

While the ability for temporal computing to improve the energy efficiency of these neural networks is intriguing, the primary temporal framework cannot directly implement these operations. Multiplication, one of the most widely used operators in neural networks, introduces significant challenges as a value in time would need to be delayed by the current time many times over. This is impractical because it would require too much time to represent results, as well as using circuitry that is difficult to implement. Additionally, temporal systems can struggle to scale, with limited memory and programmability support. Finally, training temporal neural networks can be difficult, as most of the temporal operators are non-differentiable and temporal systems have noisy circuitry that limits precision. New hardware and architectural designs are required to leverage temporal computation for neural network inference, while non-traditional training methods may be able to improve temporal neural network performance.

## Thesis Statement:

In this dissertation I show how energy efficient neural network inference can be achieved with temporal computation equipped with a transformed arithmetic and temporal data management, while energy efficient training can be enabled through hardware-driven support for Zeroth-Order optimization.

This dissertation is presented in the following manner:

- **Chapter 2** presents the negative log transformation that enables temporal arithmetic through a temporal approximation technique. It discusses the necessary hardware structures and examines the trade off between energy efficiency and operation level accuracy, while introducing recurrence, a technique for short-term temporal memory. It then presents and analyzes a near-sensor architecture for energy efficient convolutions, an important operation for vision based neural networks.

- **Chapter 3** improves upon the approximations presented in Chapter 2, then demonstrates how recurrence, coupled with devices that act as temporal memories can be leveraged to create a programmable neural network accelerator. This chapter analyzes the impact of temporal arithmetic on neural network accuracy, and presents techniques to recover this accuracy.

- **Chapter 4** introduces zeroth-order (ZO) training and finetuning. It presents and analyzes a digital accelerator for supporting ZO training to maximize data reuse and minimize additional computations. Section 4.8 discusses how ZO training can be applied to temporal neural networks, and presents some initial analysis of its impact on accuracy.

## 1.1 Permissions and Attributions

1. The content of chapter 2 is the result of a collaboration with Peiyang Song, Advait Madhavan, Jeremy Lau, and Timothy Sherwood, and has previously appeared in the proceedings of the 29th International Conferenc on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS): https://doi.org/10.1145/3620665.3640395.

2. The content of chapter 3 is the result of a collaboration with Michael Beyeler, Jeremy Lau, and Timothy Sherwood, and has previously appeared in the proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA): https://doi.org/10.1145/3695053.3731027.

3. The content of chapter 4 is the result of a collaboration with Pranjali Jain, Naomi Rehman, Eliah Reeves, Yequan Zhao, Jonathan Balkind, Zheng Zhang and Timothy Sherwood and is under review at the International Symposium on Computer Architecture (ISCA).

The pronoun used in the rest of the dissertation is *we* in recognition of my collaborators' contributions.

# Chapter 2

# Energy Efficient Convolutions With Temporal Arithmetic

## 2.1 Introduction

Convolution is a critical operation for image processing, being the basis for feature extraction [4], filters [5], and edge detection [6]. The rise of visual sensor networks [7] and convolutional neural networks (CNNs) [8] have further amplified the importance of convolutions. In the domain of sensor-embedded computation specifically, where convolution operations are typically local and resident, energy consumption is dominated by the cost of converting the data into a form that can be manipulated digitally and the cost of multiply and accumulate (MAC) operations [9].

Technology scaling helps reduce energy consumption but the slowing of traditional computational scaling, coupled with ever increasing sensor array densities, creates opportunities for non-traditional computing paradigms. While this can be as simple as reduced-precision floating point [10], more radical approaches include stochastic computing [11, 12] and race logic [13], which experiment with alternative data encodings. These

data encodings are of particular interest because they leverage the electrical behavior of basic binary logic gates in new ways. Race logic encodes information into the timing of a voltage edge and relies on four operations: minimum, maximum, (with AND and OR respectively), delay, and inhibit [13, 3] to perform computation. These operations are logically complete [14] and have a near minimal activity factor [15], creating the potential for incredibly energy efficient computation. However, to the best of our knowledge no prior work has demonstrated efficient arithmetic operations with race logic primitives, limiting the general applicability of race logic.

A primary contribution of this chapter is a new temporal value encoding — one that fully leverages the existing race-logic circuits, but that also allows for efficient arithmetic — while still representing values as a single edge.

The key insight is to apply a mathematical transformation under which addition and multiplication are co-transformed with the encoded values, forming a new mathematical ring over delays. Data values in traditional *importance space* are converted into a *delay space* with a rising edge occurring after a *delay equal to the negative log of the value*. Multiplication in importance space becomes simple addition in delay space, which means that multiplication can be implemented with a simple delay operation. Addition, however, becomes a negative log sum exponential (nLSE) function. While the nLSE function initially seems challenging to implement, we show that the function can be efficiently approximated with nothing more than min, max, and delay.

While delay space is interesting theoretically and opens many new avenues for temporal computation, it is also practical to implement, tolerant to noise, and highly energy efficient in practice. To demonstrate these points, we design and evaluate a near-sensor architecture operating completely in delay space, from sensor activation through convolution, as shown in Figure 2.1.

This proposed architecture matches the iterative row-by-row read-out of most sensor

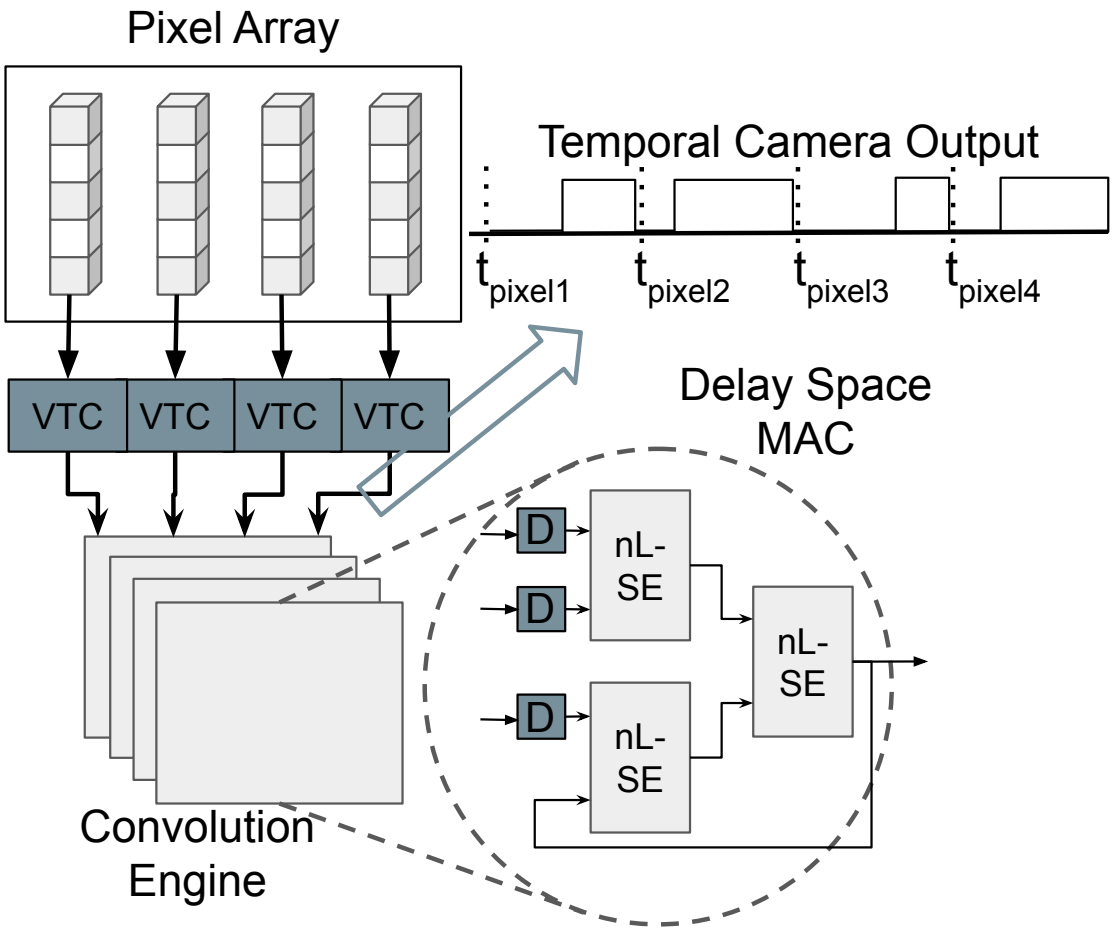



Figure 2.1: A full system diagram of our architecture. The ADCs in the staged pixel readout are replaced with voltage to time (VTC). An example of the output from one of these VTCs is shown above the system. This output is then passed to the hard-coded convolution engine. The core of this architecture is our delay space MAC where the output can be routed back to the input, creating our recurrence architecture.

arrays with a temporal scaling and summation that can be applied iteratively as each new row is read. At the core of this convolution system is an automated transformation that converts the MAC operations into nLSE and delay units in a novel recurrence architecture. This chapter presents the following contributions:

- A new data encoding that transforms the linear operations of convolution into new operations in an inverted logarithmic delay space, significantly expanding the computational potential of temporal computing.
- Efficient approximations for the new "soft" operations required in delay space, using existing "hard" temporal logic primitives, and recurrent hardware implementations of those approximations to iteratively perform delay-coded summations.
- An architectural evaluation tool for our convolution architecture and the hardware implementations of these temporal computations. We demonstrate the utility of this approach by embedding these linear operations within the natural temporal staging of pixel read-out.
- We quantify the impact of various noise factors on approximation accuracy and show that realistic designs can operate with energy efficiencies 8x better than prior work in the area.

We start with description of the encoding, its properties, and hardware implementation (Sections 2.2 and 2.3) before moving on to a more complete architecture and evaluation (Sections 2.4 and 2.5), finishing with connections to other work and final conclusions (Sections **??** and 2.6).

## 2.2 Delay Space Arithmetic

In race logic, a signal's time of arrival – *when* a rising or falling edge occurs – encodes the signal's value, rather than the signal's voltage level. Using just four basic operations on those signals, first arrival (FA), last arrival (LA), INHIBIT, and DELAY [13, 3], arbitrary temporal functions can be constructed [14]. If one thinks about the delay in time as linearly encoding a value (as has been assumed by prior work), then FA and LA execute min and max functions respectively, while the delay performs simple addition. This makes some intuitive sense as a signal computed as FA($\tau_1$,DELAY($\tau_2$, $\delta$)) will appear at time $\min(\tau_1, \tau_2 + \delta)$.

This encoding has the advantage of a very simple hardware mapping, with FA and LA on rising edges being implemented by simple OR and AND gates respectively. While not corresponding exactly to existing logic gates, INHIBIT only requires two transistors [3], and there are a myriad of hardware delay elements [16].

In addition to being a complete temporal logic, race logic has been shown to efficiently implement shortest path graph algorithms, decision trees, sorting networks, and other useful constructions [13, 3, 14]. However, efficient implementations of more general arithmetic operations with race logic primitives has never been demonstrated, in part due to the complexity of multiplication and the inability to represent negative numbers or perform subtraction. Ideally we would be able to achieve the following properties:

1. **Operations directly on encoded form**: Given a real number we need a clear way to get into and out of this encoding coupled with a way to perform the operations necessary for convolution (addition, subtraction, and multiplication) directly in the encoded form, *without* unnecessary conversions. More formally we could say we are looking for a bijective ring homomorphism of the reals.

2. **Important values early**: Traditionally, a signal of larger magnitude is represented

with a larger number. When dealing with delays, it would be far more natural for highly important values to be encoded as *shorter* durations of time, so that less important contributions can be truncated at any time. The more "excited" the system, the smaller the delay and hence the larger the importance – meaning the events that occur first are those that carry the most weight.

3. **Broader dynamic range**: From image processing to machine learning, the ability to stretch beyond the constraints of linear encoding is very useful. This is particularly critical when dealing with delays because the execution time and energy consumption is coupled to the data representation. Most prior "unary" schemes map values linearly, which makes it difficult to deal with very large and very small values in the same computation. An ideal solution would allow a broad range of values to be operated on in a way similar in spirit to a floating point representations, without introducing the problems of normalization.

We propose that a new encoding meeting all of these requirements is a negative log mapping, where a value $x$ in the original convolution maps to a signal that has a rising edge after a time delay of $x'$ where

$$x' = -\ln(x) \tag{2.1}$$

Values encoded in this way are in "delay space", and we use the notation of primes such as $x'$ to indicate delay space values. Given a delay $x'$ we can map its value back to the original "importance space" via

$$x = \frac{1}{e^{x'}} = e^{-x'} \tag{2.2}$$

All operations in importance space convolution have direct equivalents in delay space,

which can be derived by applying Equation 2.1 and logarithmic identities. A delay space multiplication can be performed using simple addition. Delay space addition and subtraction map to the negative log sum exponential (nLSE) and negative log difference exponential (nLDE) functions respectively:

$$x \cdot y \mapsto x' + y' \tag{2.3}$$

$$x + y \mapsto -\ln(e^{-x'} + e^{-y'}) = \text{nLSE}(x', y') \tag{2.4}$$

$$x - y \mapsto -\ln(e^{-x'} - e^{-y'}) = \text{nLDE}(x', y') \tag{2.5}$$

### 2.2.1 Approximate Delay Space Addition

At first glance this negative log space seems awkward – the nLSE and nLDE functions in particular seem difficult to efficiently implement in hardware. But delay space has some nice properties which we can exploit. First, as values approach infinity in importance space, they approach zero time in delay space, so more important values have less delay.

Second, because it is a logarithmic encoding, the value space gives reasonable encodings to a far wider range of values. Furthermore we can *re-scale* our values by simply shifting the reference point for the delay, because addition or subtraction in delay space is the same as multiplicative scaling in importance space. This means we have a much improved range to operate in. Finally, and most critically from an implementation standpoint, the nLSE function is a form of "soft min" operation (where LSE is the "real soft max" used commonly in machine learning [17]) and is associative, commutative, and addition distributes through it as $\text{nLSE}(a+\delta, b+\delta) = \text{nLSE}(a, b) + \delta$. These properties, along with its bounds, allow us to come up with arbitrarily tight approximations for nLSE using only min, max, and delay.

Figure 2.2 shows the nLSE function (Equation 2.4). The function is bounded from

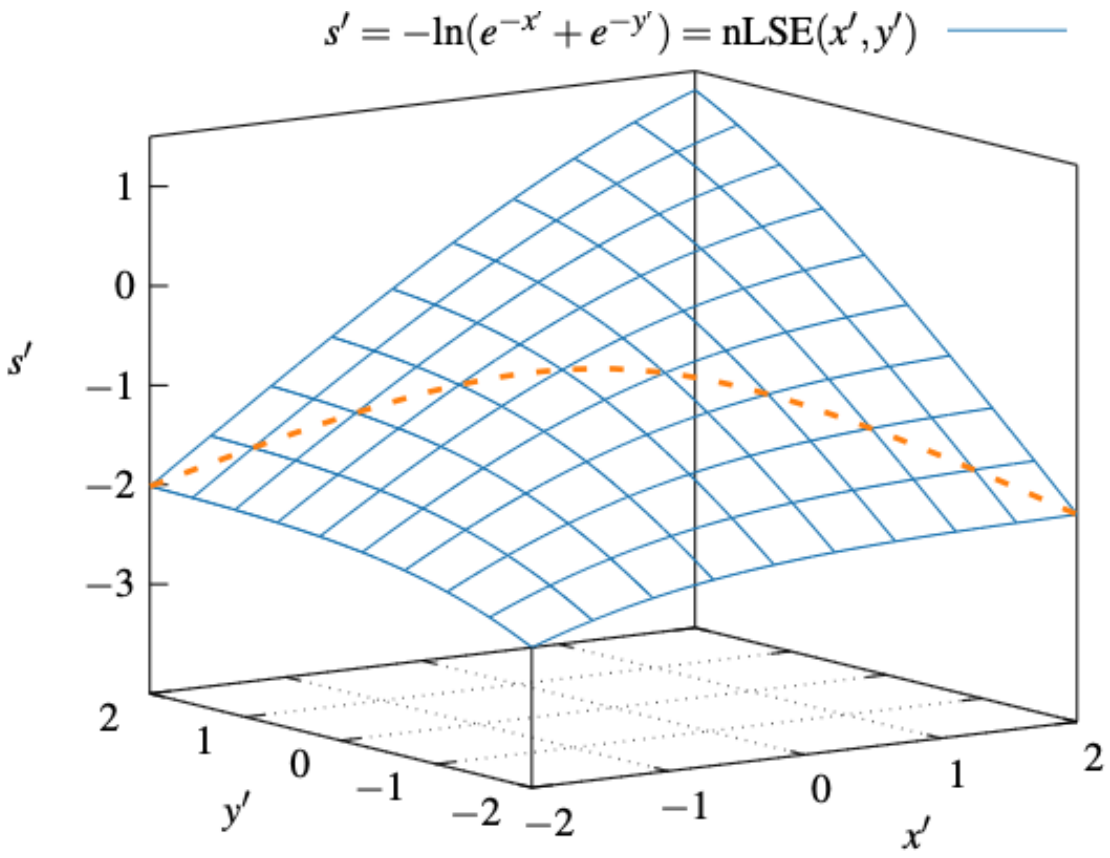



Figure 2.2: Plot of $s' = \mathrm{nLSE}(x', y')$. The surface has the same shape along all planes where $x' + y' = K$, for all constants $K$. The dashed line shows a representative slice of the surface on the plane where $x' + y' = 0$. This representative slice is shown again in Figure 2.3.

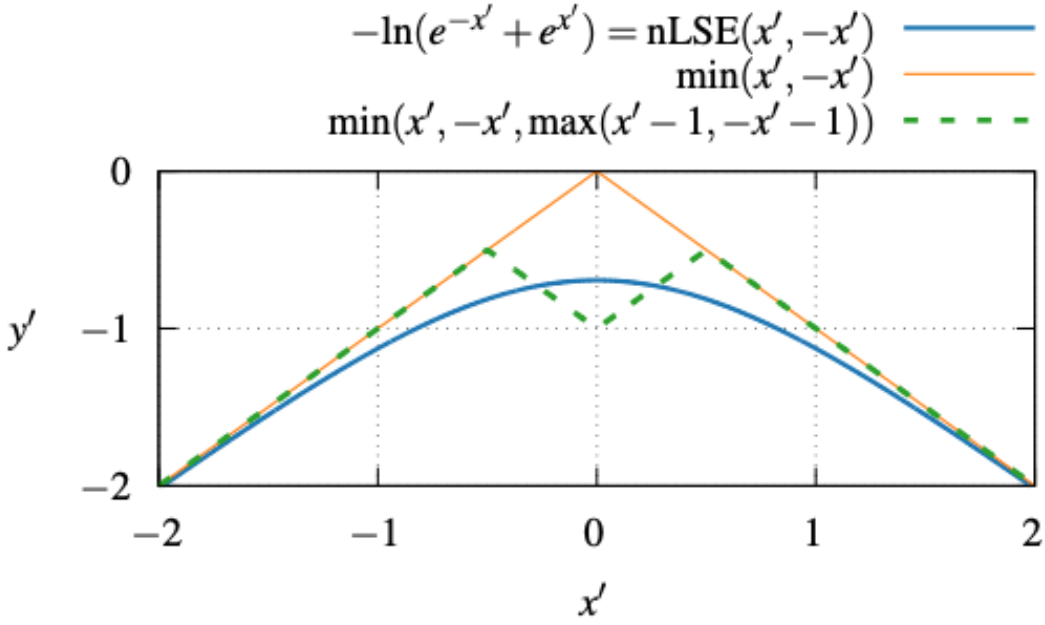



Figure 2.3: Plot of $\text{nLSE}(x', -x')$. This is the representative slice of nLSE from Figure 2.2, where $x' + y' = 0$. This graph also shows how min roughly approximates nLSE, and how introducing a max-term can improve the approximation.

above by $\min(x', y')$ and at the most extreme points, where one of $x'$ or $y'$ is much larger than the other, the behavior of $\text{nLSE}(x', y')$ converges to $\min(x', y')$. However, as $x'$ and $y'$ become closer in value, nLSE deviates further and further from min, with worst-case error $-\ln(2)$ when $x' = y'$.

While the "hard min" of $\min(x', y')$ can serve as an approximation of nLSE, introducing a shifted max-term $(\max(x' + C, y' + D))$ under the min adds a valley to the approximation, which can reduce approximation error. An arbitrary number of these max-terms can be added, and approximation error can be made arbitrarily small with

more max-terms. An approximation with $n$ max-terms has the form:

$$\begin{aligned} &\min(x', y', \max(x' + C_0, y' + D_0), \\ &\quad \dots \\ &\quad \max(x' + C_{n-1}, y' + D_{n-1})) \end{aligned} \tag{2.6}$$

To improve our nLSE approximation with max-terms, we must find appropriate values for the constants $C_i$ and $D_i$. To do so, we first observe that the curves of nLSE$(x', y')$ in Figure 2.2 all have the same shape for any arbitrary constant $K = x' + y'$. The dotted line in Figure 2.2 shows a representative slice of this surface, where $x' + y' = 0$. All slices parallel to this representative slice have the same shape. Therefore, without loss of generality, we can set $K = 0$ and focus on approximating nLSE$(x', -x')$. This observation allows us to simplify the problem from two inputs $x', y'$ to one input $x'$. Figure 2.3 shows nLSE simplified to one input, how min roughly approximates nLSE, and an improved approximation with one max-term, where $C_0 = D_0 = -1$.

Figure 2.3 reveals that this simplified one-input nLSE is symmetric over the $y'$-axis, so it is sufficient to approximate only non-negative $x'$ values. If negative $x'$ values are involved, we can negate the inputs and thus simplify the calculation down to non-negative nLSE. This same trick works for two-input nLSE, which can be computed after swapping $x'$ and $y'$ if $x' < y'$, keeping the first operand always greater.

We implement the min-of-max approximation model from Equation 2.6 in the Pyomo [18] modeling framework. The model is generalized to represent approximations with any number of max-terms, so we model several approximations with various numbers of max-terms, and use the KNITRO solver [19] to optimize each approximation's curve fit. Figure 2.4 shows a sample optimized approximation with four max-terms. In Section 2.5.2 we show that seven max-terms is more than enough to achieve reasonable

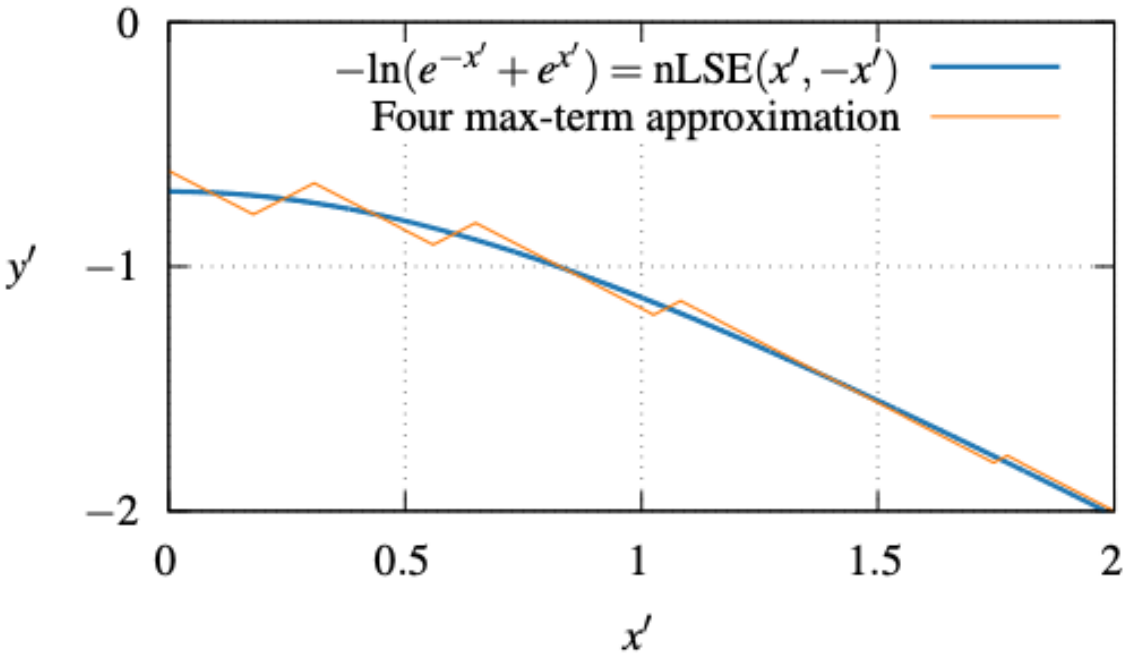



Figure 2.4: Approximating nLSE with four max-terms. We only approximate positive $x'$ values because Figure 2.3 is symmetric about the $y'$-axis.

accuracy.

### 2.2.2 Negative Numbers and Subtraction

While addition and multiplication give us most of what we need to perform convolution, an additive inverse is required to handle negative constants common to many convolution kernels and complete the mathematical ring. To enable subtraction and negative value representation we adopt an approach similar to memristive crossbars [20] or dual rail computing [21], where we split all numbers into non-negative pairs, $\langle x_{pos}, x_{neg} \rangle$. If the value is positive, $x_{pos}$ equals the value and $x_{neg}$ is 0. For a negative value, $x_{pos}$ is 0 and $x_{neg}$ is the absolute value. For zero, both $x_{pos}$ and $x_{neg}$ are zero.

With this representation, subtraction simply becomes addition with the $x_{neg}$ field of the second operand, but we must re-normalize the result to ensure that at least one of

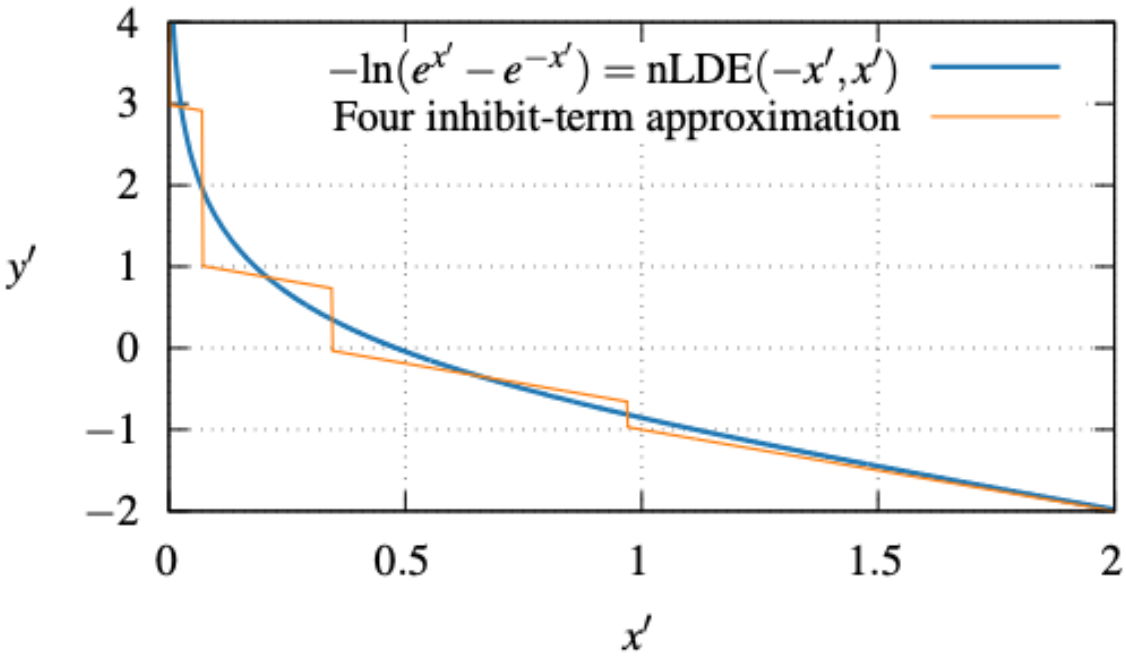



Figure 2.5: Approximating nLDE with four inhibit-terms. nLDE's shape is more difficult to approximate than nLSE's shape, because nLDE converges to infinity as $x'$ approaches 0, while nLSE converges to $-\ln(2)$ as $x'$ approaches 0 (see Figure 2.4).

$\langle x_{pos}, x_{neg} \rangle$ is zero. This re-normalization only has to occur at the end of computation and once per convolution. In importance space this re-normalization is achieved with subtraction:

$$x_{pos} = \begin{cases} x_{pos} - x_{neg} & \text{if } x_{pos} \geq x_{neg} \\ 0 & \text{if } x_{pos} < x_{neg} \end{cases}$$

$$x_{neg} = \begin{cases} x_{neg} - x_{pos} & \text{if } x_{neg} \geq x_{pos} \\ 0 & \text{if } x_{neg} < x_{pos} \end{cases}$$

In delay space we re-normalize with the negative log difference exponential function nLDE (Equation 2.5), which is also symmetric about the $y'$-axis. The nLDE function is bounded by the inhibit function, similar to how the nLSE function is bounded by

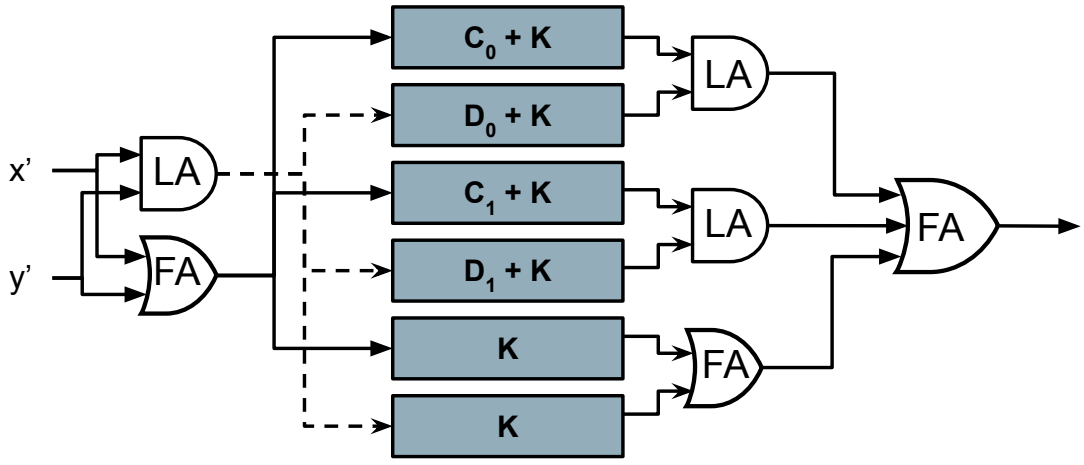
min. inhibit$(t_i, t_d)$ accepts two inputs, an inhibiting event set at time $t_i$ and a data event arriving at time $t_d$, and outputs an event at time $t_d$ if and only if $t_d < t_i$. When $t_d \geq t_i$, no event will be output, which is equivalent to an event at time $\infty$. We use min-of-inhibit functions to approximate nLDE:

$$\begin{aligned} &\min(\text{inhibit}(x' + E_0, y' + F_0), \\ &\quad \text{inhibit}(x' + E_1, y' + F_1), \\ &\quad \dots \\ &\quad \text{inhibit}(x' + E_{n-1}, y' + F_{n-1})) \end{aligned} \tag{2.7}$$

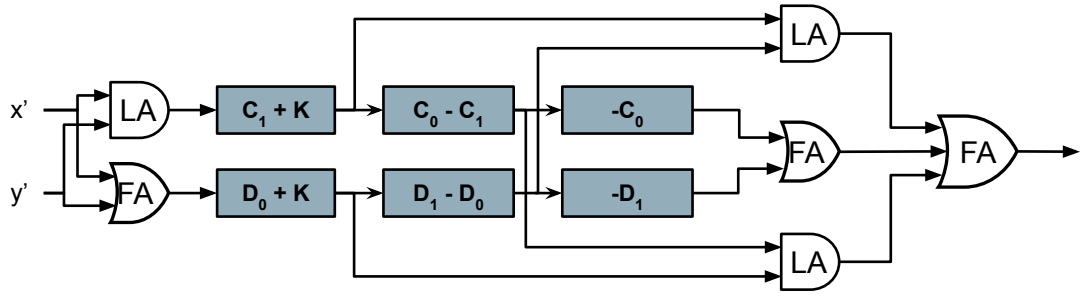
Using the same technique as the nLSE approximation, we simplify from two inputs to one, modeling min-of-inhibit approximations from Equation 2.7 where $x' + y' = 0$ in Pyomo, and use KNITRO to optimize approximations with various numbers of inhibit-terms. Figure 2.5 shows a sample optimized nLDE approximation with four inhibit-terms.

### 2.2.3 Approximation Logic Design

One of the challenges with a delay-based nLSE function is that its output is always less than or equal to the min of the inputs. When mapped to times, this would mean the output needs to fire before any of the inputs have even arrived and thus the constants in equation 2.6 are all negative. Instead of directly implementing nLSE, we implement a time shifted nLSE (which is the same as rescaling in importance space). This can be done by adding a constant $K$ that is greater than or equal to the most negative approximation




(a)




(b)

Figure 2.6: Figure (a) shows A naive circuit implementation of a 2 max-term nLSE approximation where all of the max-terms are calculated in parallel. Figure (b) gives an Optimized hardware implementation of a 2 max-term nLSE approximation using two chains of delay elements. Since the $C_i$ and $D_i$ constants are in opposite order (the largest $C_i$ pairs with the smallest $D_i$) the output of the first delay element is matched with the last delay element. The grey blocks represent fixed delay elements.

constant:

$$\begin{aligned}
&\text{nLSE}(x', y') + K \\
\approx &\min(x', y', \max(x' + C_0, y' + D_0), \ldots) + K \\
= &\min(x' + K, y' + K, \max(x' + C_0 + K, y' + D_0 + K), \ldots)
\end{aligned}$$

The now-positive max-term constants can now be implemented with constant delays, and the max and min functions can be replaced with LA and FA gates respectively. Finally, to handle swapping $x'$ and $y'$, as described in Section 2.2.1, we use a temporal comparator circuit [14] at the input to ensure proper ordering, preventing the need to double number of max-terms. A naive implementation of this can be seen in Figure 2.6a where each max-term has its own dedicated delay.

However, this approach creates redundant delay, wasting both energy and area. With physical delay elements, energy consumption scales linearly with the magnitude of delay, so designs should minimize the number of redundant delays. This informs an optimized design shown in Figure 2.6b, where there is only a single path of delay elements for each input. The max-term inputs are tapped from the proper location along the delay chain. Note that as max-terms are added, the $C_i$ constants increase and the $D_i$ constants decrease, so max-terms are connected in reverse order along their respective chains. This same shift-and-chain approach can be used to create a hardware implementation of the nLDE approximation function (Equation 2.7).

## 2.3 Recurrence Architecture

The accumulation operation in multiply-accumulate implies many connected summations. This can be achieved by all inputs arriving at the same time and being summed

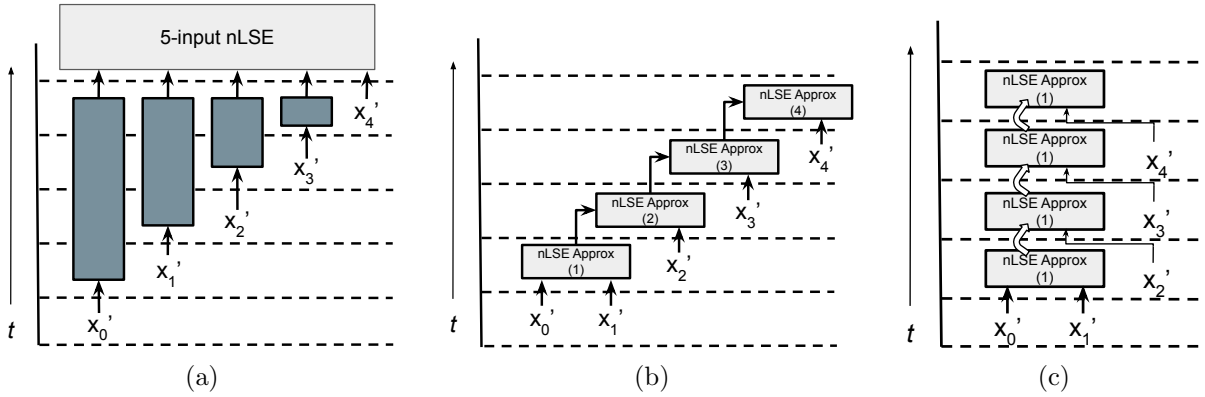



Figure 2.7: Illustrations of different approaches to reference frame synchronization, where the $y$-axis indicates time and the $x$-axis indicates physical area added to the design. Figure (a) shows how each element can be individually delayed to ensure the proper reference frame. Figure (b) shows how computation can be performed as inputs arrive, and Figure (c) shows how the output of this computation can be looped to the same hardware (as shown by the hollow arrow) block with some delay to prevent hardware replication.

together. However, for most systems this is infeasible due to either the number of inputs or data naturally arriving over time. Therefore, a stateful system is required, which poses a problem for delay space. Temporal memories have been proposed in the past but they all rely on either emerging technologies or complex and sensitive analog circuits[22, 23, 24, 25]. To solve this problem we propose an alternative race logic approach that emulates a stateful system for efficient MACs.

This approach relies on the fact that in temporal systems, the arrival time of any event, $t_x$, is meaningless without context. There must also be a reference time $t_{ref}$, where the difference between the times $t_x - t_{ref}$ represents $x$'s value. All inputs to race logic operations, including our nLSE approximations, must share the same reference time to function properly. In a system with fully parallelized inputs this is simple with a shared global reference time $t_{ref} = t_{start}$. However, with serialized inputs each input will have a unique local reference time. Before the data can be processed together with race logic it

must be synchronized to the last input's reference time. One method of synchronization is delaying each input by the difference between their local reference time and the last input's reference time: $t_{x'} = t_x + (t_{last} - t_{local})$, where an example using nLSE is shown in Figure 2.7a. These delay lines create state by holding in-flight data until all other inputs have arrived.

This represents an important trick where we can maintain the proper *logical* value of a signal with a shifting reference frame by adding a constant (delay). However, this requires as many delay lines as inputs, which wastes energy and area, making it infeasible for large systems. Instead, we propose an optimization that operates on inputs as they arrive, performing as many operations as possible on currently available data. Then, only the required information for the next stage of computation is passed, minimizing the data that must be reference shifted. This also allows the required reference shifting delay to be reduced because each operation has some inherent delay, either through explicit delay elements or gate propagation time.

Our nLSE approximations are particularly well suited for this compute-on-arrival approach because 1) they have a larger delay due to the addition of a constant $K$, even further reducing the required frame shift delay, and 2) only accept two inputs. Figure 2.7b illustrates this by showing how a large nLSE operation can be split up and staged across input arrival times. The equivalence of breaking apart the nLSE operations can be shown simply:

$$\begin{aligned} -\ln(e^{-x} + e^{-y} + e^{-z}) &= -\ln(e^{\ln(e^{-x}+e^{-y})} + e^{-z}) \\ &= -\ln(e^{-\mathrm{nLSE}(x,y)} + e^{-z}) \\ &= \mathrm{nLSE}(\mathrm{nLSE}(x,y), z) \end{aligned}$$

When the calculation being performed is completely associative, commutative and

symmetric then each stage is composed of identical hardware, such as our nLSE approximation in Figure 2.7b. When these conditions are satisfied then the compute-on-arrival approach can be further optimized by frame shifting the output and looping it back into its own input, as shown in Figure 2.7c. In order for this to operate properly, three constraints must be met:

- The rise and fall time of the delay elements must be matched so that the integrity of the voltage pulse is maintained.
- The value of any input cannot be so large that it extends past the next value's reference frame.
- A relaxation period must be introduced between cycles to ensure that the previous cycle's falling edge does not interfere with the computation of the current cycle.

This *recurrence* takes advantage of reference shifting delay lines and causes the system to act like a classical state machine, so long as the inputs arrive at evenly spaced time intervals. This saves both energy and area by limiting the number and length of delay elements, as well as reducing the number of nLSE approximation hardware blocks required. It's worth noting that this approach is not restricted to a fully serial input. The two-input nLSE approximation units can be expanded to a tree of nLSE blocks with any number of inputs. Regardless of the size of the tree, the output can still be recurrent, creating a trade-off between the number of nLSE approximations, the area of the design, the minimum length of each cycle, and the number of necessary cycles.

## 2.4 Rolling Shutter Convolution Architecture

Most cameras rely on a technique called a rolling shutter [26] where individual columns or rows of pixels are captured and read out in parallel. This pipelines both the exposure

time and the ADC readout times while causing the inputs to become available across fixed time intervals. This acts as a natural serialization for our recurrence architecture and can be leveraged for convolution. Even in systems without a rolling shutter, a staged readout is often applied to reduce the number of necessary ADCs.

### 2.4.1 Analog to Delay Space

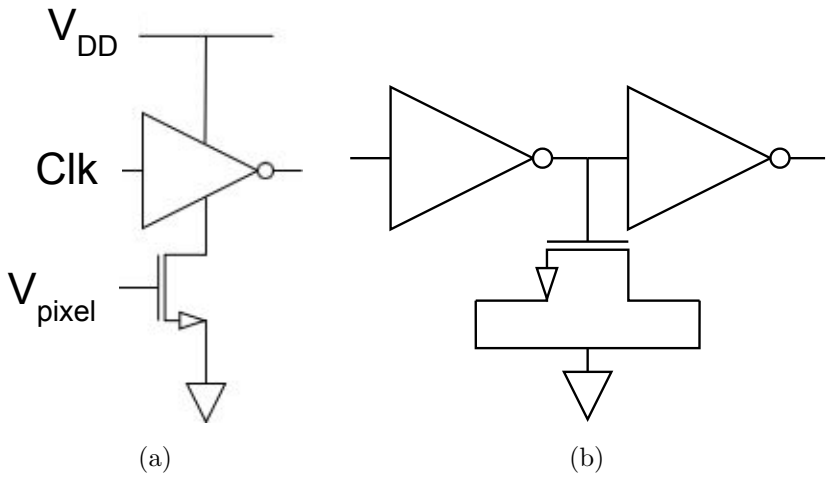



Figure 2.8: Figure (a) Demonstrates a simple starved inverter that forms the basis of voltage to time converters (VTC). Figure (b) Shows the circuit diagram of an inverter-chain delay element with a transistor between the inverter output and ground to increase the delay of a single inverter.

Traditional camera systems rely on Analog-to-Digital Conversion (ADC) circuits to read pixel data prior to processing. However, low power systems have explored converting analog signals into the temporal domain [27, 28] for mixed signal processing. This approach has shown the potential for significant energy savings [29], and allows for temporal processing of data while in the time domain.

Many of these techniques attempt to achieve a linear mapping of input voltage to output timing [27, 30]. However, this is insufficient for our delay space computation as the negative log of the input is required. Instead, we require a voltage to time converter

(VTC) that matches the negative log properties of delay space. This can be achieved by using a starved inverter, shown in Figure 2.8a, which is already at the core of many VTC systems [29].

The pixel voltage controls the current starving transistor, and the read clock acts as the input to the base inverter. As the pixel voltage increases, the delay of the inverter decreases, creating the value inversion necessary for delay space. Additionally, pixel voltage has a monotonically increasing impact on the delay of the clock signal, allowing it to approximate negative log for specific regions of interest. These two properties combined allow for the delay of the clock signal to be interpreted as a delay space value.

### 2.4.2 Delay Elements

Three previous approaches to delay elements for race logic have been proposed: discrete delay through chained DFFs [13], inverter chains, and starved inverters [15]. These approaches all have their advantages, but the discretized approach would be impractical to integrate with the precise constant values required for the delay space approximations.

The starved inverter approach surrounds an inverter with two statically biased transistors [15]. These inverters act as current sources to reduce the rise and fall time of the output, resulting in controllable delays based on the bias voltages. However, this approach requires each starved inverter to be biased separately, introducing circuit complexity and additional energy consumption. Also, having a single large delay element maximizes the impact of random jitter (RJ) from a single element.

Conversely, an inverter chain uses a large number of identical inverters, with the effective load capacitance determining the delay from each individual inverter. In this approach the RJ for each element is independent and scales with the magnitude of delay, causing each additional element to reduce the overall impact of RJ [31]. In our design

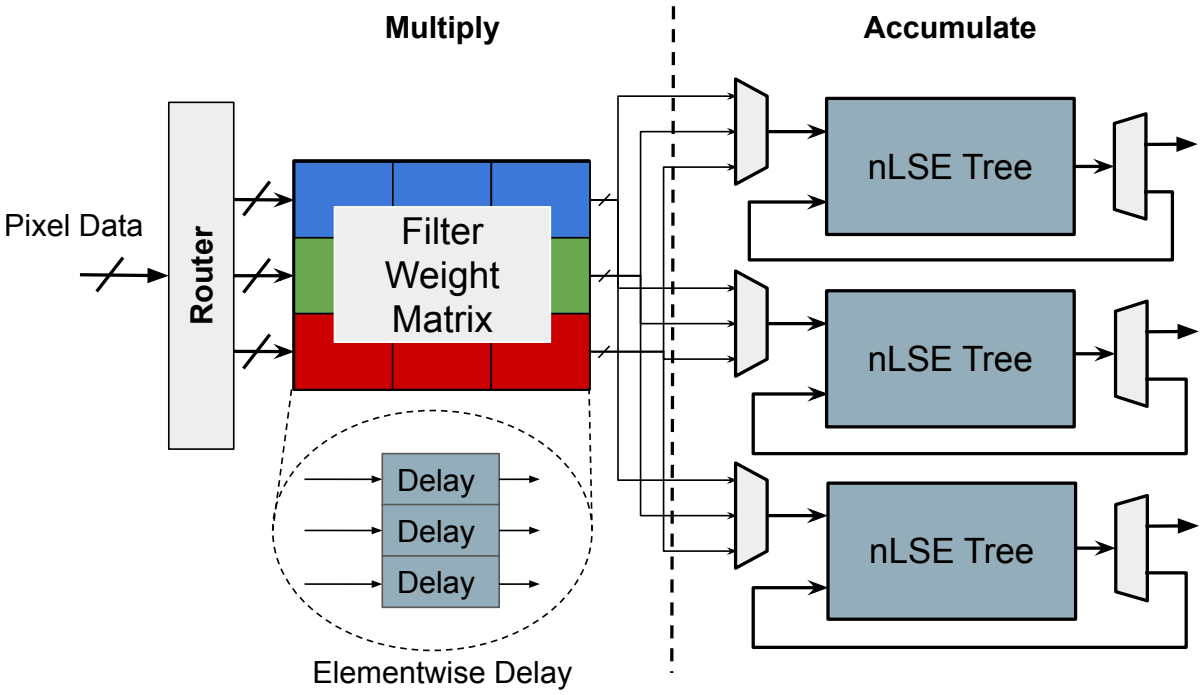
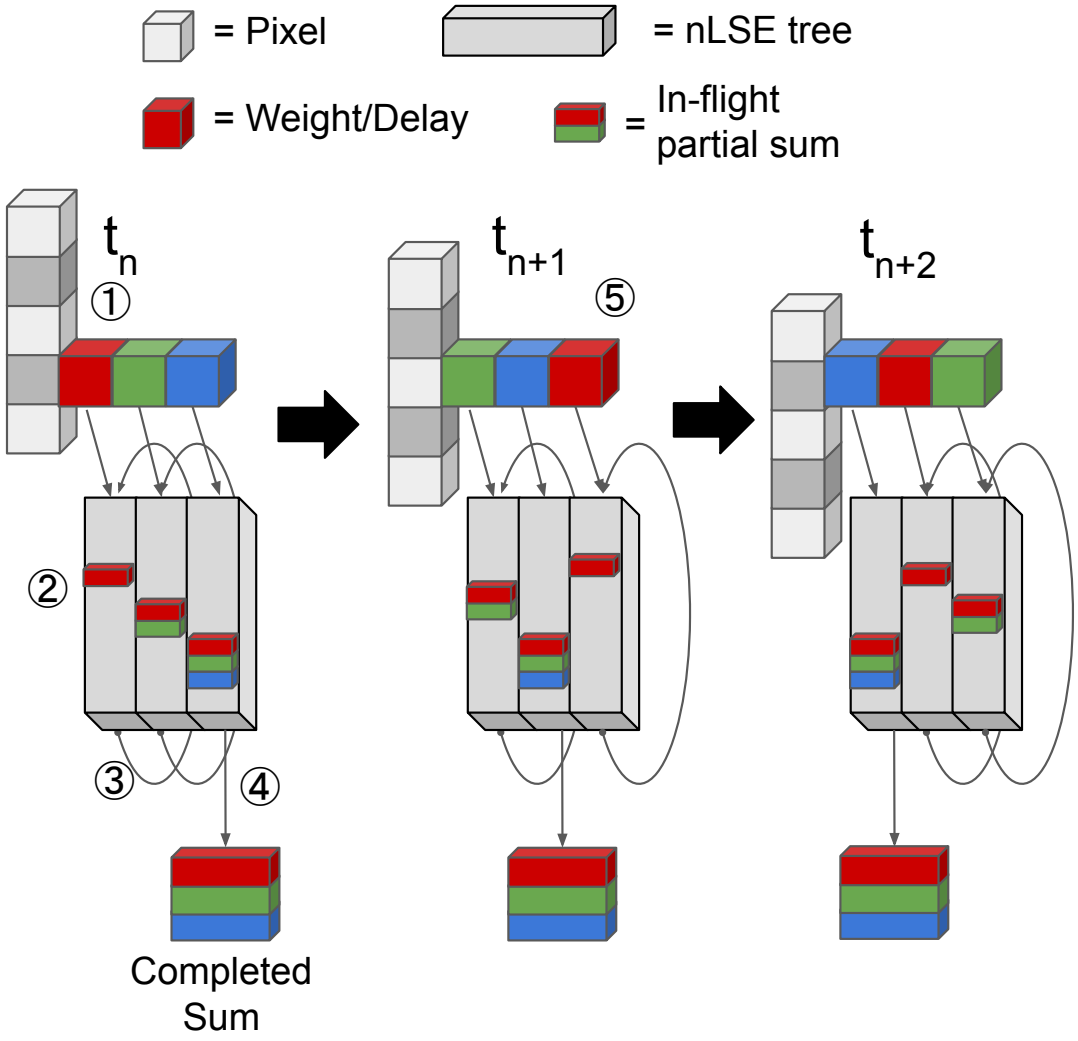



Figure 2.9: A single, hard-coded, three by three convolution filter. Data is read through the rolling shutter, then progresses through the dedicated filter weight delay matrix and finally to the accumulation nLSE trees. The non-recurrent outputs from each nLSE tree is combined for a single output. The multiply and accumulate hardware handles one row of convolutions and must be replicated for each necessary row.

we use this approach and hard-code the load capacitance by adding a transistor between the output of each inverter and ground as shown in Figure 2.8b. The load capacitance is varied by changing the size of the ground transistor, allowing delay to be hard-coded into the hardware. This chained inverter approach introduces an area and noise trade-off, as the larger the output capacitance, the fewer inverters are required for each chain. We evaluate how RJ impacts the accuracy of our approximations in section 2.5.2, which informs the design of the ground transistor.

Figure 2.10: An example of the scheduled pixel information flow through the system for a filter with 3 rows. Each differently colored cube represents a different *row* in the filter weight matrix. In subsequent time steps each nLSE tree receives inputs from next filter row until it has completed a full filter application. The actions corresponding to the circled digits are explained in Section 2.4.3.



### 2.4.3 Dedicated Convolution Engine

The core of this architecture is a hard-coded convolution MAC block shown in Figure 2.9 which handles all of the convolutions along a set of columns equal to the filter width. This block must be replicated for every filter application along the row axis of the given pixel array, given by $1 + \frac{pixelwidth - filterwidth}{stride}$.

Once the pixel values have been converted to delay space through the VTCs, each temporal pixel value is passed to all MAC blocks that utilize it. Figure 2.10 illustrates how this pixel data flows through the MAC block. Each block has a static matrix of delay lines that includes every filter weight. As the corresponding pixels are processed by the VTCs they are passed to this matrix and the input can be distributed to any or all of the filter's rows. The number of rows activated in a given cycle is given by $\lceil filterlength/stride \rceil$, where a stride of 1 indicates that every filter row will be used each cycle. ① Each activated filter row then performs element-wise delay, which functions as the multiplication of the MAC operation.

To fully utilize the outputs from the activated rows there must an accumulation unit for every potentially activated filter row. ② These accumulation units receive the results of the filter rows and subsequently sums the incoming values in delay space. The core of the accumulation unit is a tree of two-input nLSE approximation units performing delay space addition. Whenever the tree is not fully symmetric, gaps in the tree must be path balanced by with a delay equal to the delay of the nLSE approximation hardware to maintain the proper reference frame. This delay is added as deep in the tree as possible to minimize the number of extra delay lines. When an image begins to be processed only one accumulation unit is activated, with the next one being activated every *stride* cycles. Eventually they will all be activated and operating in parallel, similar to a systolic array.

③ The result of this nLSE tree is then looped back as the recurrence discussed in

Section 2.3. This loop must have a delay equal to the cycle time minus the inherent delay of the nLSE tree. ④ Once an nLSE tree has received an input from each of the filter rows it passes its result to the output instead of being looped for recurrence. Then, in the next cycle ⑤ the outputs of the filter rows are rotated so that each nLSE receives the next necessary input for their application of the filter. This ensures that there will be an output produced every *stride* cycles.

### 2.4.4 Split Value Representation

The split value representation presented in section 2.2.2 requires each MAC block to have multiple kernels to handle each combination of {positive, negative} {input, weight}. The temporal outputs of the VTCs are always non-negative, so we only need two kernels (positive input, positive weight and positive input, negative weight). If the inputs could be negative then we would need a total of four kernels to handle the two additional cases.

Since the weights are hard-coded, the filter weight matrix is split between the positive and negative sides. Weights that are zero in importance space, either because of the pos/neg split or the actual value is zero, become infinity. An infinite delay is the same as the path not existing, allowing the number of weight multiplications (delays) to stay constant with filter size, unaffected by the split value representation. Additionally, the number of nLSE approximations performed remains the same, the only extra operations are any required tree balancing and the recurrence delay lines.

Once a convolution have been completed, both positive and negative kernels must be routed to a delay space subtraction unit to re-normalize the values as described in Section 2.2.2. The subtraction unit evaluates an nLDE approximation (Section 2.2.3) on two inputs, and the result can be fed directly to the output. This output can either remain in delay space for further temporal computation, or be converted to the digital

domain for traditional processing.

## 2.5 Evaluation and Results

### 2.5.1 Architectural Simulator

To explore the architectural space we created an architectural simulator that takes a system description as its input and produces a software representation of the architecture. The system description includes the image dimensions, the kernel shape, the number of kernels and the convolution stride. This architecture can be configured to change the unit scale, the maximum supply voltage swing, and the magnitude of each inverter's delay (as a multiplier of the minimum inverter delay). We use the unit scale to indicate the connection between theoretical delay values and physical time values: for example an abstract delay of 1 could map to 5ns.

These parameters are then used to estimate the area and energy consumption for the given system description. The energy estimates are based on SPICE simulations for delay lines using 65nm predictive technology models [32]. Area estimates are dependent on typical transistor sizes for 65nm nodes and an estimation for the total number of transistors in the system. We assume that the delay elements dominate both the energy and area and assume that the control logic is negligible. The architectural parameters are also used to implement noisy versions of delay and our approximations based on noise values from [31].

The generated architecture can be executed given an image data set and filter. The convolution is performed according to the compiled architecture with programmable multiplication, addition and subtraction functions. We use these programmable functions to ensure that when using importance space operations the architecture produces the exact

same result as software convolution. We also verify that using exact delay, nLSE and nLDE provides the same result as software convolution when the results are converted from delay space to importance space.

### 2.5.2 Approximation Accuracy and Noise

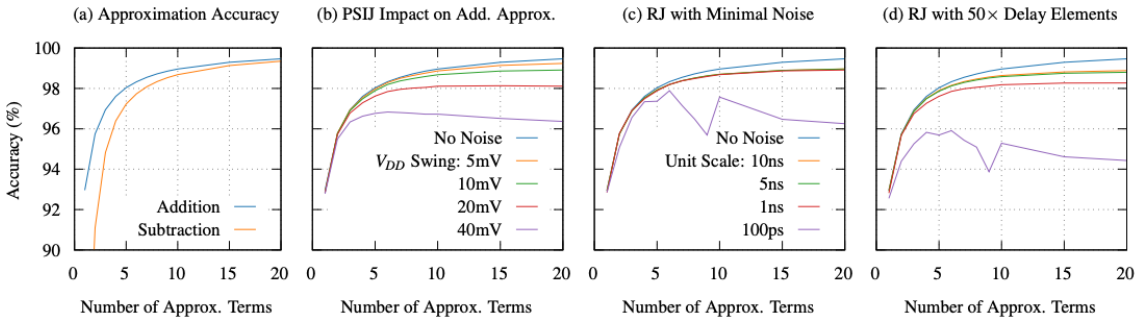



Figure 2.11: Delay space approximation accuracy as the number of approximation terms increases. Figure (a) shows the pure approximation accuracy, while Figure (b) shows how PSIJ impacts accuracy. Figure (c) and figure (d) show how RJ impacts accuracy. All graphs share the same $y$-axis, and (c) and (d) share the same legend. The top line is the same across all graphs, showing delay space approximation accuracy for addition, without noise.

To evaluate our approximation accuracy, we generate two uniform random values between zero and one, which correspond to positive values in delay space. The values are converted to delay space, the approximation is applied, and the result is then converted back to importance space. This approximation result is then compared to the exact operation being performed in importance space. We perform this operation a million times, then take the range-normalized RMS error (RMSE) to determine the overall accuracy.

We use this approach to evaluate how the number of approximation terms impacts the accuracy of our nLSE and nLDE approximations with infinite precision, shown in Figure 2.11a. The graph shows that additional approximation terms significantly increase the approximation accuracy until 7 or 8 terms where they start to provide diminishing returns. However, when implemented in hardware, the accuracy of the approximation

is also impacted by hardware timing noise. We use the noisy approximation simulator described in Section 2.5.1 to evaluate the two major sources of noise: power supply induced jitter (PSIJ) and random jitter (RJ) [31]. PSIJ is a product of the power delivery network (PDN) and will dominate the noise unless the swing in the supply voltage is carefully controlled. Figure 2.11b shows how the accuracy of our nLSE approximation suffers due to the noise introduced by varying $V_{DD}$ swings.

While this shows that PSIJ can significantly reduce the accuracy of the approximations, the low power nature of our convolution architecture puts less stress on the PDN, reducing potential voltage swings. If this is still insufficient to control the PSIJ, the voltage swing can be further controlled by adding decoupling capacitors to the PDN. RJ, on the other hand, cannot be controlled and is a function of the magnitude of each inverter's delay [31]. However, since the RJ of each inverter is independent, the more inverters in a chain, the smaller the impact to the system. This creates a relationship between the delay of each inverter, the unit scale, and noise.

Figures 2.11c and 2.11d shows the impact of limited PSIJ (10mV $V_{DD}$ swing) and RJ for different unit scales with fixed delay element magnitudes. Figure 2.11c uses the smallest possible delay for each inverter, minimizing RJ, while Figure 2.11d uses an inverter with 50× the minimal delay. Both of these show that there is a minimum unit scaling that must be met to ensure the max-terms can be utilized fully, otherwise the RJ noise dominates. With appropriate unit scaling and minimal noise there is little to no impact on the accuracy, but this requires significantly longer inverter chains to achieve the proper delay. However, Figure 2.11d shows that for a unit scale of 5ns the size of the inverter chains can be cut by 50× with minimal noise impact. Beyond 5ns the hardware approximation improves slightly, but may not justify the extra energy required for the longer unit scaling.

Surprisingly the impact of noise, regardless of source, is larger as more approximation

terms are added. Each additional term reduces the difference between approximation constants ($C_i$ and $D_i$ in Section 2.2.1), which increases the probability that the FA gate selects the wrong approximation term due to noise. The incorrect approximation term will then have a larger impact on the output than slight variations along the proper max-term.

Note that the nLDE approximation is also affected by noise, but because there is a larger difference between its approximation constants, the noise impacts the accuracy to a lesser degree. Due to space constraints we omit the nLDE noise trade-off graphs from this paper, but we consider this impact on nLDE accuracy in our architectural evaluation.

### 2.5.3 Architectural Evaluation

To investigate the relationship between approximation accuracy and the our convolution architecture, we run a design space exploration with our architectural simulator. For our exploration we use the Imagenette [33] dataset, which is a subset of the Imagenet [34] dataset, and scale each image to 150 by 150 pixels. Our architecture is then configured to run the Sobol function from OpenCV [35] which uses two $3 \times 3$ filters. We sweep the number of approximation terms for both nLSE and nLDE as well as the unit scale. The delay of each inverter is set to $50\times$ the minimal delay and the maximum $V_{DD}$ swing is set to 10mV. For each configuration, the architectural simulator determines the energy consumption. Then it emulates all of the operations with appropriate noise in the same order as the simulated hardware for a single channel of the dataset. The Sobel convolution was calculated for five different images and the RMSE for each output pixel was taken. The results from this architectural explorations are shown in Figure 2.12.

The first noteworthy result from this graph is the effect of unit scale. Each vertical grouping represents a unique unit scale, caused by the necessary increase in delay

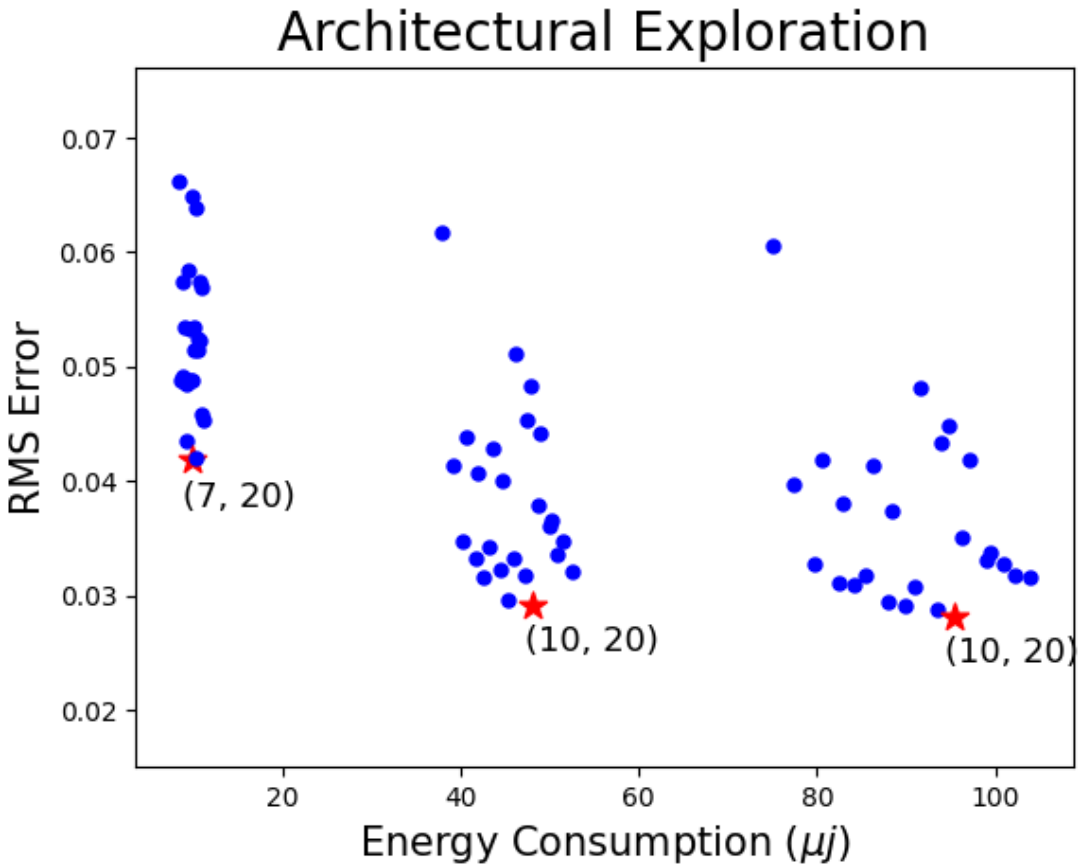



Figure 2.12: The results of the architectural exploration where the number of approximation terms are swept across 5, 7, 10, 15, 20 for both nLSE and nLDE and the unit scale is swept across 1ns, 5ns and 10ns. Circled points are along the Pareto optimal frontier, with the corresponding configuration (Unit scale, nLSE terms, nLDE terms) written next to it.

Table 2.1: Convolution benchmarks and their descriptions. The filter configuration is (filter size, stride, number of filters).

| **Function** | **Description** | **Filter Config** |
|---|---|---|
| Sobel | Edge Detection | $3 \times 3$, 1, 2 |
| pyrDown | Blur and Downsample | $5 \times 5$, 2, 1 |
| GaussianBlur | Blur with Gaussian filter | $7 \times 7$, 1, 1 |

magnitudes, which leads to a larger energy consumption. Also, the energy difference between different approximation configurations becomes more significant as the unit scale increases. This is because the increased delay due to additional approximation terms is multiplied by the larger unit scale, leading to more energy consumption.

As expected, the accuracy increases noticeably when going from 1ns to 5ns unit scales. However, similar to the two input accuracy in Figure 2.11, the difference between 5ns and 10ns is fairly small. This demonstrates an effect where increasing the unit scale improves the accuracy with diminishing gains while the energy continues to increase consistently. It is clear from the Pareto frontier that maximizing the number of inhibit terms is important for accuracy. As discussed earlier in Section 2.5.2, inhibit terms are less affected by noise. Also, adding additional inhibit terms leads to a smaller increase in the energy consumption compared to adding max-terms because there are significantly fewer nLDE operations. Due to the larger impact of noise on the nLSE approximation, increasing the number of max-terms beyond seven or ten does not significantly improve the accuracy.

In Figure 2.12 we highlight three configurations that are along the Pareto optimal frontier and use these configurations to investigate the functions shown in Table 2.1 from the openCV library [35]. Note that the pyrDown and GaussianBlur functions have non-negative weight values, so the split value representation and nLDE approximation unnecessary. Table 2.2 shows the area, energy and cycle time of these functions for each of the highlighted configurations. The Sobel function is the only one with negative weights,

Table 2.2: Relevant statistics for different race logic convolution configurations for three different functions. The Arch column indicates the unit scale, the number of max-terms and the number of inhibit-terms. The pyrDown and GaussianBlur functions do not have inhibit-terms as they have only positive filter weights. Throughput is measure in millions of frames per second (Mfps)

| Function | Arch. | Area ($mm^2$) | Energy per Frame ($\mu J$) | Max Through-put ($Mfps$) | Acc. ($RMSE$) |
|---|---|---|---|---|---|
| Sobel | 1ns, 7, 20 | .02 | 9.81 | 71 | .065 |
| | 5ns, 10, 20 | .08 | 48.1 | 18 | .029 |
| | 10ns, 10, 20 | .149 | 95.4 | 9 | .028 |
| pyrDown | 1ns, 7 | .004 | 7.2 | 55 | .038 |
| | 5ns, 10 | .134 | 36.6 | 12 | .029 |
| | 10ns, 10 | .236 | 72.7 | 6 | .028 |
| GaussianBlur | 1ns, 7 | .008 | 14.2 | 55 | .037 |
| | 5ns, 10 | .273 | 73.1 | 12 | .028 |
| | 10ns, 10 | .481 | 146 | 6 | .027 |

Table 2.3: Comparison between the delay space convolutions and a state of the art, processing in pixel convolution engine. We show the energy of the operation, the delay of each operation, the accuracy and the energy delay product. The delay space convolution analysis includes the energy cost of the VTC, and separate columns are given for a system that includes the TDC cost. The minimum energy delay product for each configuration is shown in bold.

| Benchmark | | PIP | | | | Delay Space | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Shape | Stride | Energy per pixel per frame ($pJ$) | Frame Delay ($mS$) | E×D Product ($pJ \times mS$) | Error ($\%RMSE$) | Energy per pixel per frame ($pJ$) | Energy w/TDC ($pJ$) | Min Frame Delay ($mS$) | E×D Product ($pJ \times mS$) | E×D Product w/TDC ($pJ \times mS$) | Error ($\%RMSE$) |
| 2x2 | 2 | 16.9 | 12.8 | 2.18e2 | 7.18 | 16.4 | 21.9 | 7.35e-4 | **1.21e-2** | 1.61e-2 | 3.69 |
| 2x2 | 4 | 4.6 | 5.2 | 2.39e1 | 7.12 | 4.2 | 9.8 | 7.35e-4 | **3.13e-3** | 7.2e-3 | 3.51 |
| 2x4 | 2 | 32.9 | 21.9 | 7.21e2 | 7.8 | 21.3 | 26.8 | 7.35e-4 | **1.57e-2** | 1.97e-2 | 3.02 |
| 2x4 | 4 | 7.0 | 7.7 | 5.42e1 | 6.77 | 5.46 | 11.0 | 7.35e-4 | **4.01e-3** | 8.09e-3 | 3.6 |
| 4x4 | 2 | 104 | 41.3 | 4.29e3 | 4.56 | 41.0 | 46.6 | 1.47e-3 | **6.04e-2** | 6.86e-2 | 2.8 |
| 4x4 | 4 | 11.6 | 1.3 | 1.52e2 | 5.27 | 10.3 | 15.9 | 1.47e-3 | **1.52e-2** | 2.34e-2 | 3.2 |

requiring the split value representation. However, it still consumes a similar amount of resources to pyrDown due to the fact that it has significantly smaller filter sizes. While pyrDown consumes marginally less energy than the Sobel function, this is largely due to the fact that its stride size is two, effectively halving the number of necessary computations and static hardware. GaussianBlur demonstrates that larger filter sizes consume significantly more resources, doubling the energy and area requirements from pyrDown while barely increasing the accuracy. However, because the height of the nLSE tree is the same for both functions, the maximum possible throughput is the same.

The maximum possible throughput for all of these architectures is incredibly high, with the fastest filter able to process 71 million frames-per-second (FPS), although it is unlikely that a camera would able to match this throughput. So in a slower camera the delay space architecture will be able process the convolutions without slowing down the throughput of the system. The only requirement is that the row readout cycle time be evenly spaced and short enough for recursion be energy efficient. If this means the readout is faster than the rest of the camera operations then the time between finishing an image and starting to read the next can be lengthened to match the rest of the system's FPS.

This shows how delay space convolutions can be used for a variety of workloads with low energy and no impact to the processing speed. To see how this impact compares to a state of the art approach we consider a processing-in-pixel (PIP) architecture [36] as shown in Table 2.3. For our comparison we evaluate the benchmark used in the PIP paper, a 1.5 bit edge detection convolution where the weights can be zero, 1 or -1. We evaluated our design against three different filter sizes with two different strides. For the calculation of our energy we include both the VTC [30] and TDC [37] cost, and use a 1ns unit scale, 10 max-terms and 20 inhibit-terms delay space configuration. We report the energy per pixel per frame as the same figure of merit presented by the PIP work. We also ran the same convolution benchmark as the PIP in our simulator, and normalized our RMSE to the range of output values to create a percent error for direct comparison.

The delay space approach consumes less energy than PIP for every convolution structure when the results are left in the temporal domain. As the convolution gets larger and the stride stays small, the energy improvements of the delay space architecture becomes more significant, achieving over 2.5× energy savings for a four by four filter with a stride of two. The delay space architecture also outperforms PIP when the stride size is small, but PIP improves its energy consumption more with an increasing stride size than delay space. This makes delay space a better fit for most openCV filtering applications, where the stride size normally always small. However, when the results must be converted into digital the delay space architecture begins to consume more energy for the ultra-low energy design points. This motivates additional computation in the temporal domain, such as more convolutional layers or min/max selections.

Also, the delay space architecture approach can be completely separated from the pixel array, unlike PIP. PIP requires circuitry to be added to the pixel array, significantly reducing the possible pixel density and the speed of image processing. This causes the energy delay product of the PIP architecture to be several orders of magnitude higher

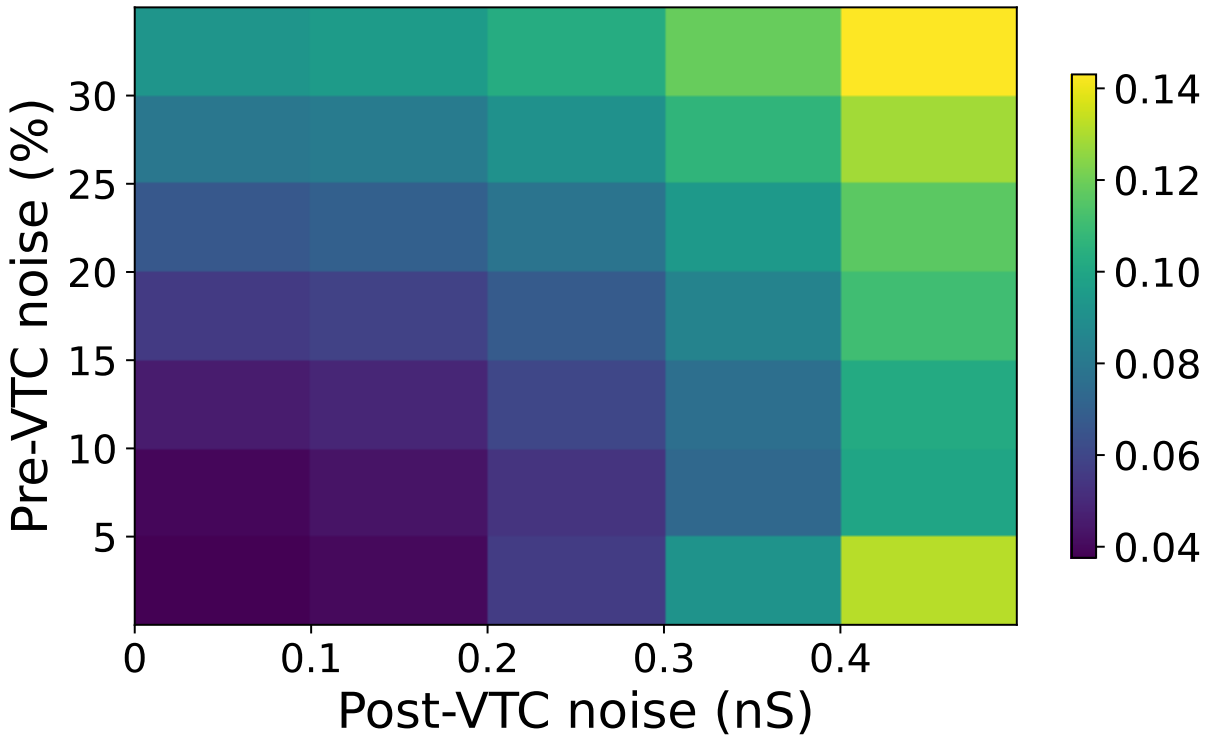



Figure 2.13: Impact of sensor and VTC noise on the accuracy of the pyrDown convolution. The axes show the standard deviation of the introduced error while the color indicates the output RMSE. The $y$-axis is given as a percentage of the max possible range of input values while the $x$-axis is given in nanoseconds.

than the delay space architecture, which operates fast enough that it has minimal impact on the image throughput.

### 2.5.4 VTC noise impact

We've shown that our proposed delay space arithmetic is able to process convolutions with incredibly low energy while still maintaining good accuracy, but this evaluation assumes a perfect sensor and VTC. In reality, these systems are noisy, which can contribute to error in the final result of the delay space convolutions. This noise falls into two general categories, noise that occurs before the VTC and noise at the VTC output. Noise that occurs before the VTC are associated with the CMOS sensor, such as fixed pattern

noise [38] and dark shot noise [39], and affects the voltage used to program the starved inverter. The resulting value will then undergo a negative log transformation with the conversion to delay space. Noise that occurs after the VTC is caused by nonidealities in the VTC, and results in a linear change in the delay space value.

To evaluate the impact of these two types of noise we use the pyrDown application from OpenCV [35]. We use the same testing framework and dataset as before, but instead we add a random Gaussian distribution to each image centered around zero at two different points, before the negative log conversion to delay space, and after. For both types of noise we gradually increase the standard deviation of the distribution. The noise is applied in simulation, and represents an abstract noise model to show how robust the system is to noise. For the delay space arithmetic we use a 1ns unit scale, a 10 max-term system with a maximum $V_{DD}$ swing of 10mV and the same noise model as before. The results of this experiment are shown in Figure 2.13. The $y$-axis indicates noise on the original, importance space image, where the values are the standard deviation of the noise, shown as a percentage of the maximum input range. The $x$-axis shows the noise applied after the conversion to delay space, where each value is the standard deviation of the noise in nanoseconds. The color or each "pixel" corresponds to the output RMSE from an ideal convolution with no error and noise.

As expected, the RMSE increases with the noise being added to the system. However, the error of the delay space computation grows slower than the amount of noise. With a noise distribution that has a standard deviation of 10% of the input range and no post-VTC noise, the RMSE is .046, less than .01 greater than the baseline which assumes a perfect input. Also, the post-VTC noise has a small impact until it reaches a standard deviation of .3 ns. This is because the post-VTC noise is added in the log domain, which means it has an exponential impact on the importance space input. So small errors have very little impact, but as the post-VTC noise increases the error increases exponentially.

This exponential growth is most apparent in the case when there is no pre-VTC noise. The RMSE grows faster along this row than any others, and has almost as much error as the maximum noise data point. This is because both noise sources are added independently, so the two sources of noise can partially counteract each other. However, in a real system there will always be some noise both before and after the VTC. Overall this chart shows that the delay space architecture is fairly robust to noisy inputs and can tolerate some non-idealities caused by the VTC.

## 2.6 Conclusion

There is no question that convolution operations will continue to play an important role in sensor information processing, with applications from image processing to object detection and neural networks. The power consumption of multiply-accumulate (MAC) operations is a key factor in convolutions integrated with sensor operation. Departing from the norm of performing full analog-to-digital conversions for each pixel, we showed how to perform arithmetic on temporally encoded signals with remarkable energy efficiency.

At the heart of our approach is a negative log transformation, converting the traditional numeric space into a 'delay space'. This mechanism enables multiplicative scaling by adding delays. We demonstrate the direct execution of negative log-space addition and subtraction in this new delay space, ensuring that normal associative and communicative properties of addition still apply in the transformed operations. Moreover, we show how strong approximations of these operations can be efficiently constructed from delay elements and existing CMOS logic elements.

Many computations execute iteratively, and to apply temporal techniques we need new techniques for chaining and recurrently operating these designs, multiplexing in both

time and space. We show how time-division multiplexing aligns naturally with the staged ADC readout common to most modern sensor arrays. To establish the practicality of this approach, we present an automated transformation that carries traditional convolutions through to their delay space equivalents. This translation balances error introduced by a new temporal equivalent of quantization and delay element noise. We use this approach to show how our approach can consume eight times less energy than another state-of-the-art convolution approach while achieving similar accuracy. We believe that this approach presents a powerful new set of design primitives with applications beyond convolution.

# Chapter 3

# Single Spike Artificial Neural Networks

## 3.1 Introduction

Time is an inherent property of all physical actions, one that is increasingly leveraged by designers. Devices that can sense, interact, and make decisions about the physical world around them are increasingly in demand [40], and one of the simplest ways to perform these tasks is by utilizing their temporal dynamics. Analog sensors first convert their value to temporal wavefronts, then to digital [41], while other sensors such as dynamic vision cameras [42] and lidar [43] already produce outputs with information encoded in time. This motivates a simple question: can we leverage these temporal dynamics without performing expensive time to digital conversions (TDCs)? This becomes an increasingly pressing question as we drive neural networks to energy constrained devices to create systems that not only develop an understanding of the world, but also make decisions based on that knowledge.

One approach for extracting these dynamics is spiking neural networks (SNNs), which

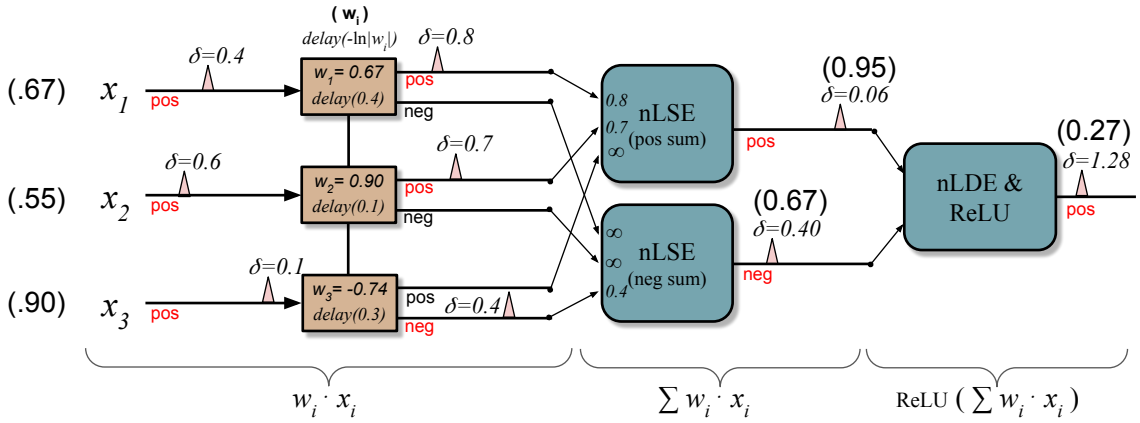



Figure 3.1: An example of a dual rail delay space neuron, with multiplication replaced by delays, accumulation performed with negative log sum exponential (nLSE) functions, and activation replaced by negative log sum difference (nLDE). The values on the far left show the original input and weight values before the delay space conversion. Each intermediate signal's delay space value is indicated with $\delta$, with earlier values appearing towards the right. Higher importance values have lower delay and are thus first to arrive at downstream units. All weight delays are converted, quantized and stored digitally, with each weight's sign indicating whether to route the delayed value to a positive or negative rail (shown in red). Each value in the computation is represented by a single "spike" represented as a rising edge on a wire.

leverage neurons that extract spatial and temporal patterns from a series of spikes occurring across time. This is a seemingly completely different paradigm from artificial neural networks (ANNs) (where linear layers of multiply-accumulate operations are interspersed with non-linear activation layers) and SNN's bio-inspired nature points towards the potential for incredibly energy efficient inference. The potential for energy efficiency comes largely from the data representation, in which spikes are encoded as binary values that facilitate lightweight, simple operations. However, this may also come at the expense of accuracy [44], latency [45], and design time [46]. Meanwhile, ANNs benefit from decades of research and industrial efforts creating mature tools and techniques that have shown to be effective across the computing stack, providing highly effective networks in terms of both accuracy and performance. In this chapter we propose a new approach that merges these seemingly distinct forms of inference, which can leverage the sparse, temporal cod-

ing of SNNs while still implementing the same algorithms underlying ANNs to achieve new levels of energy efficient inference. When coupled with traditional digital memories, this approach performs well compared to the most efficient and custom-trained SNNs, and when integrated with memristive temporal memories [47], will even beat 8-bit digital logic in terms of energy efficiency without any retraining or fine-tuning.

The key to our approach is the insight that we can encode each activation of the ANNs into just a single "spike" in time, relying on the temporal operations of "delay" and "real softmin" (operations that are both well defined and easily implementable on sets of spikes) to compute the linear operations that dominate ANN computation: multiplication and accumulation. Under this scheme, weights are applied to input spikes by simply delaying their propagation in time, as shown in Figure 3.1, which corresponds *exactly to multiplication*. A large weight in the neural network corresponds, very intuitively, with a small delay — with a zero weight equivalent to an infinite delay. Summation, under the same transformation, becomes an efficiently computable function using the primitive operations inherent to the temporal domain [2]. In this work we show how several important functions for ANN inference can be implemented using temporal primitives, creating a unified framework for implementing ANNs using a single spike per activation.

As one attempts to apply this arithmetic at larger scales, two new problems arise: a) previous demonstrations of real softmin approximations show some critical non-idealities leading to error accumulation and b) internal digital memory becomes a greater and greater limiter. To address the first problem, we propose a noise-tolerant approximation method that can recover the accuracy of the original ANN through temporally interleaved min-term chaining. To address the second, we show the potential of temporal memories — memristive crossbars that have been demonstrated to capture and replay delayed signals — to bring the energy consumption below an 8-bit digital ANN. The resulting systolic array relies on programmable delay elements to provide a form of temporal

quantization, while using temporal recurrence [48, 1] to maintain values in the temporal domain, reducing the number of memory accesses and facilitating classical dataflows. The specific contributions of this chapter are:

- We describe how delay space arithmetic and approximations implement a spiking version of the linear operations in ANNs, then demonstrate how this method can be extended to efficiently implement ReLU and max-pooling.
- We are the first to evaluate how previously proposed delay space approximations impact neural network accuracy, and propose a new approximation methodology that provides improved network accuracy and better noise resiliency.
- We propose a hybrid temporal/digital systolic array that allows for weights to be loaded digitally, yet perform all operations in the temporal domain. Then we analyze how different architectural parameters impact network accuracy.
- We demonstrate how our systolic array can directly integrate with temporal memory systems, creating a system that consumes 3.5× less energy than a digital systolic array.

## 3.2 Neural Networks with Delay Space

SNNs attempt to model the relationship between presynaptic and postsynaptic spikes in a way that is biologically plausible. This is generally done with a leaky-integrate-and-fire model [49], which integrates incoming spikes and fires after reaching an activation threshold. These spiking models exhibit a variety of behaviors and may be abstracted further based on how information is encoded into the spikes — typically into one of two general categories, rate-coded and temporal-coded.

| Importance | Delay Space |
|---|---|
| $x$ | $-\ln(x) = x'$ |
| $w \cdot x$ | $-\ln(w \cdot x) = -\ln(w) + -\ln(x) = w' + x'$ |
| $x + y$ | $-\ln(e^{-\ln(x)} + e^{-\ln(y)}) = -\ln(e^{x'} + e^{y'}) = \text{nLSE}(x', y')$ |
| $x - y$ | $-\ln(e^{-\ln(x)} - e^{-\ln(y)}) = -\ln(e^{x'} - e^{y'}) = \text{nLDE}(x', y')$ |

Table 3.1: Traditional "importance space" operations and what they correspond to in delay space. At any point a spike delay of x' from reference time can be interpreted as an importance of $x = e^{-x'}$.

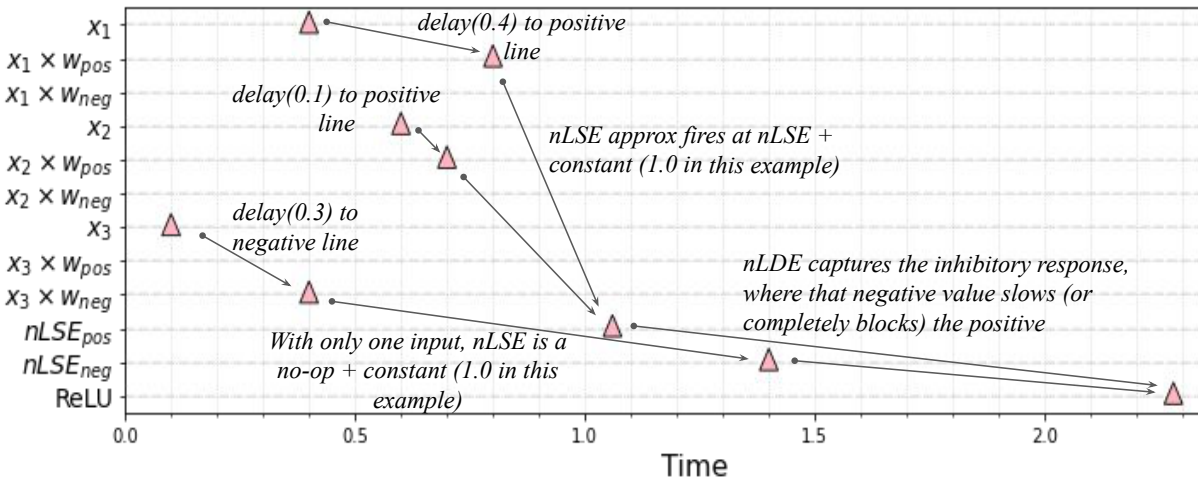



Figure 3.2: Graph showing the time of arrival and arithmetic dependencies for temporal spikes with the same example data from Figure 3.1. This additionally shows the constant offset required by the nLSE operation.

Temporal coding has inspired a general, logically complete algebra with four primary operations: min (first arrival), max (last arrival), delay (addition of a constant), and inhibit [50]. These operations have been shown to be incredibly energy-efficient when each "spike" is encoded as the rising edge of a voltage [2], giving each operation a simple logical equivalent (e.g. min corresponds to an AND). This temporal coding has been shown to be effective in decision trees [3], but there remains a large gap between the current capabilities of temporal computing and modern neural networks.

Delay space arithmetic [48] is based on this logically complete algebra, but relies on a negative natural log of the numbers and operations. Since the logarithmic function is undefined for negative numbers it necessitates a dual rail encoding to represent the complete number line. A value $x$ is split and becomes $x_{\text{pos}}, x_{\text{neg}}$, where $x_{\text{pos}} - x_{\text{neg}} = x$, $\max(x_{\text{pos}}, x_{\text{neg}}) = |x|$, and $\min(x_{\text{pos}}, x_{\text{neg}}) = 0$. This pos/neg split cannot be directly converted to delay space, as one will always be zero, and the log of zero is undefined. To simplify the transition we define the delay space equivalent of zero to be infinity instead of undefined. Infinity in temporal coding is the same as a wire that never transitions from low to high. This means one of $x'_{\text{pos}}$ or $x'_{\text{neg}}$ will always be an inactive wire.

This transformation creates a natural fit for the four primary operations of temporal computing. Table 3.1 details how standard arithmetic, or "importance space", operations become transformed in the new negative log domain. Multiplication in importance space becomes addition in delay space, which can be implemented temporally with a simple delay. This is shown on the first functional block on the left side of Figure 3.1. With an assumption of positive only inputs (which we discuss in Section 3.2.1), this delay only needs to be coupled with a router to a positive or negative rail, depending on the sign of the weight. Next, addition in importance space becomes the negative log sum exponential (nLSE) function. This function, known as real-soft min can be approximated by a min-of-maxes approach, efficiently implemented using delays and basic logic gates,

shown in Figure 3.3. The key technique that allows this to work is adding a constant offset to the result of the computation to avoid having to trigger an event before any inputs arrive, demonstrated in Figure 3.2. This accumulation in a dual rail system is shown as the middle functional block of Figure 3.1.

The result of this multiplication and accumulation will be maintained with two rails, as the negative and positive rails cannot be accumulated using the same function. However, these two rails both carry a value, breaking our original definition of dual rail coding. In importance space a simple subtraction can normalize these values, which gets transformed in the negative log domain to negative log difference exponential (nLDE). nLDE can be approximated using a min-of-inhibits, similar to the nLSE approximation. However, this function, like the original negative log transformation, is undefined for negative values. Instead it must be applied twice, with the inputs reversed:

$$y_{\text{pos}} = nLDE(x_{\text{pos}}, x_{\text{neg}})$$
$$y_{\text{neg}} = nLDE(x_{\text{neg}}, x_{\text{pos}})$$

The approximation returns $\infty$ when nLDE is undefined, so the result of these two operations will always adhere to the original definition of the dual rail coding. In order to maintain this strict dual rail definition this normalization must be applied after every operation. However, the associative nature of the preceding linear operations (additions and multiplication) apply to their log transformation (nLSE and addition), so the nLDE only needs to be applied before non-linear operations.

These three transformed operations (delay, nLSE, and nLDE) *exactly* implement the linear operations within neural networks.

### 3.2.1 Non-Linear Temporal Operations

Unlike prior work, we show this framework can be extended to several other operations to support the non-linear activations necessary for ANNs, an integral part of neural networks. The first important operation for energy-efficient neural networks is ReLU, where only positive values are passed through and all negative values become zero. In delay space this zero becomes infinity, or a wire that never triggers. Therefore, to implement ReLU, only the positive rail of the previous computation needs to be calculated. Due to the inhibit function used in the nLDE approximation, the required infinity is the natural result when the result would be negative in importance space. This makes the min-of-inhibits approximation a natural fit, as digital implementations of nLDE would return undefined or NaN.

The second operation necessary in many convolutional neural networks (CNNs) is max-pooling. If one of the values in the max-pool operation is positive, then a simple min of the positive rails can implement max-pooling. This max to min conversion stems from the negative log transform, where larger values in importance space are transformed to smaller delays in delay space. Alternatively, if none of the values are positive, then a max of the negative rails is used. This can be simplified to:

$$y_{\text{pos}} = \min(x^0_{\text{pos}}, ..., x^n_{\text{pos}})$$

$$y_{\text{neg}} = \max(x^0_{\text{neg}}, ..., x^n_{\text{neg}})$$

The simplification of this logic works because any positive value will have an infinite negative rail, causing the max of all negative values to never trigger. For networks where ReLU is combined with max pooling, the operation becomes even more simple. The only relevant information is the positive rail and is performed simply with a min operation. These two operations are sufficient to implement many neural networks [51], and pre-

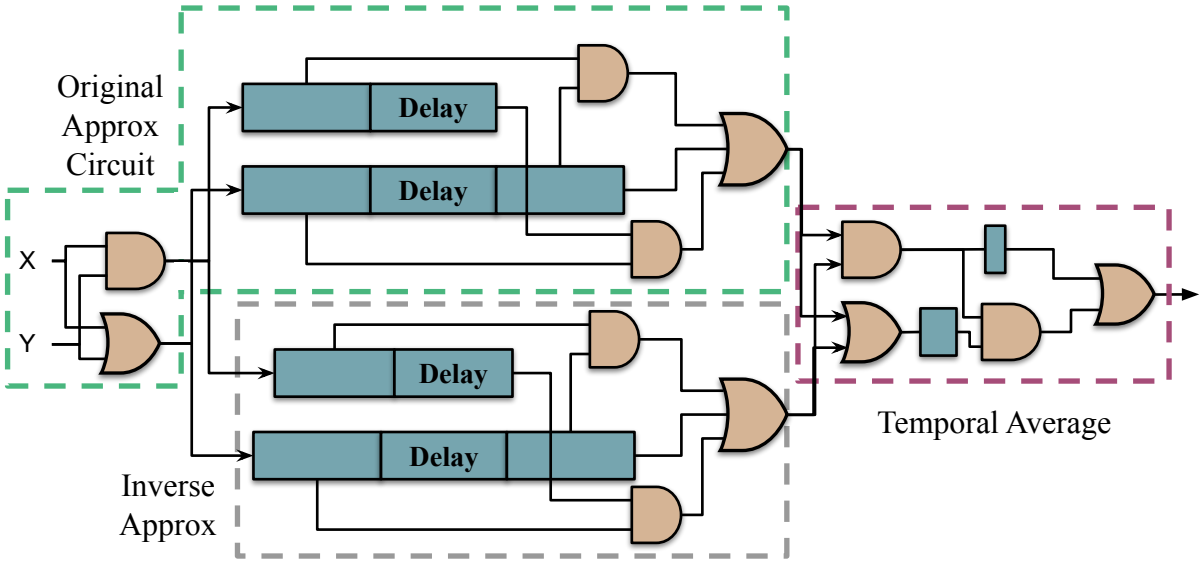



Figure 3.3: Hardware diagram of the improved nLSE approximation circuit, relying on redundancy to provide better accuracy and noise characteristics.

trained networks can easily be converted to delay space by simply transforming all inputs and weights to delay space values. For the weights of a network, this represents a one-time cost that can be performed before loading them onto the system. For the inputs, this step can be co-located with the input pre-processing, and only needs to be performed once before the network begins, representing a tiny fraction of the overall computational demand of the network.

### 3.2.2 Noise Resilience

We've shown that we can algorithmically merge ANNs with temporal computation, however, while temporal computation relies on digital tools and building blocks, the domain is inherently noisy. Further, the log transformation that enables temporal arithmetic introduces logarithmic bias, where balanced error and noise in delay space become skewed in importance space. In Section 3.6.1, we show how this degradation requires significant area and timing overheads when naively applying delay space approximations

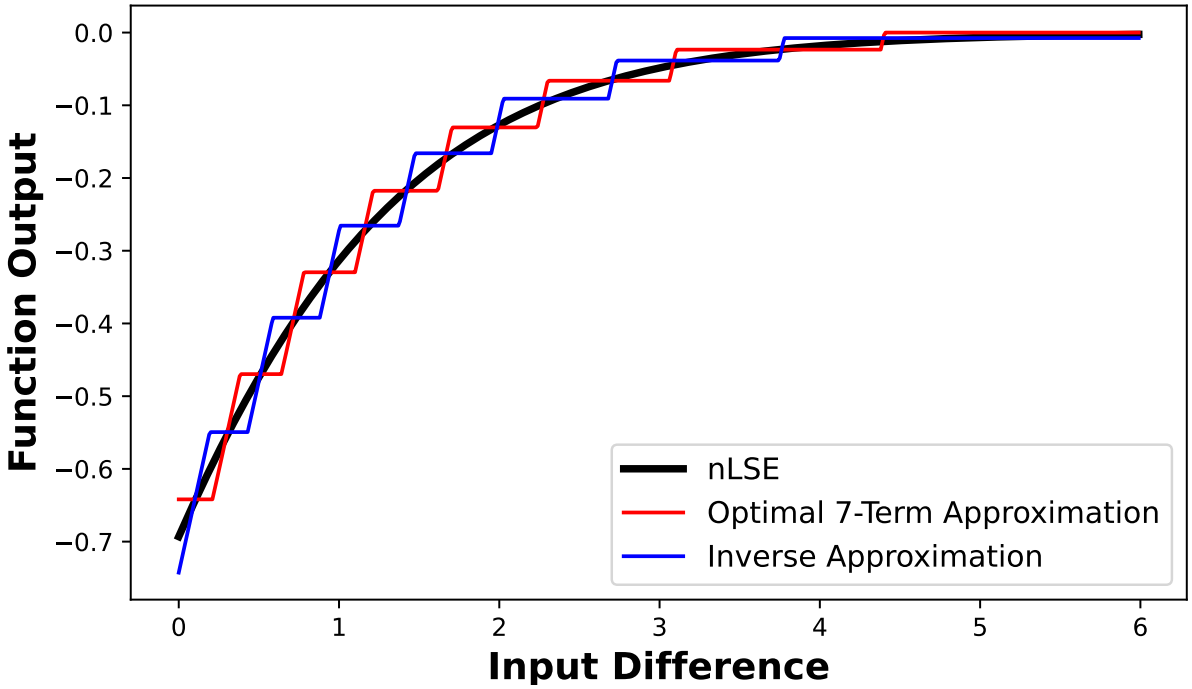



Figure 3.4: Function graph of nLSE and 2 complementary min-of-max methods of approximation.

to neural networks. To reduce the impact of both the noisy temporal operations and the approximation bias, we discuss two techniques. First, we propose a novel, circuit level optimization of the nLSE approximation that reduces logarithmic bias while simultaneously being more resilient to noise. Then, we discuss how standard regularization techniques create networks that are more robust to the types of noise introduced by our temporal networks.

**Improved approximations:** Approximations of delay space addition (nLSE) rely on a min-of-maxes approach, where every max-term creates a deformation, or "valley", in the original min function. These valleys create lines with positive and negative unit slopes, and careful placement of these valleys allows for the non-linear function nLSE to be approximated. Additionally, this approach only needs to approximate half of the function's range by operating on the difference between the inputs and adding the results to the minimum input. This is a common trick when computing real soft min

and max, and can be implemented in hardware using a simple temporal routing scheme shown on the left side of Figure 3.3. This changes the function being approximated to $\text{nLSE}(0, \max(x, y) - \min(x, y))$, which is shown as the black line in Figure 3.4.

The red line in Figure 3.4 shows a 7-term approximation using the same optimization method as [48] to place each valley. This method attempts to minimize the distance between the approximation and the original function by balancing the area above and below the original curve. However, as we show in Section 3.6.1, this approach struggles to fully recover the accuracy of convolutional neural networks. This is largely because error that's balanced in log space becomes unbalanced when converted back to importance space, introducing an inherent bias to the approximations.

Our proposed solution to this bias relies on creating an equal, but complementary approximation to the original, shown in Figure 3.4 as the blue line. This approximation uses the same optimization framework as the prior work [48], but starts the solver using a valley instead of a peak. Taking the average of these two approximations provides a near-exact copy of the original nLSE function. Unfortunately, implementing an *exact* temporal average would require analog circuitry, preventing the temporal logic from being implemented with purely digital building blocks. Instead, an approximate averaging circuit can be built by characterizing the differences between the approximations.

This approximate average circuit is developed by using the delay space equivalent to the average of two values: $\frac{x+y}{2}$ becomes $\text{nLSE}(x', y') + \ln 2$. This allows the same nLSE approximation technique to be used, with all constants shifted by $\ln 2$. However, the characterization of the two inverse approximations provides a tight bound on the difference between the approximations at any given point along the curve. Now only a small portion of the function's curve needs to be approximated, allowing an acceptable approximation with minimal additional delay and gates.

This combined approximation, shown in Figure 3.3, creates a better piecewise ap-

proximation of the original nLSE function, providing better accuracy than the original approach. Temporally combined approximations additionally provide redundancy, allowing the resulting value to be less sensitive to noise along either of the lines.

**Neural Network Regularization:** Regularization is commonly used when training neural networks to prevent over-fitting. There's a broad range of techniques, weight decay [52], noisy layers [53], and dropout [54], which attempt to prevent networks from becoming overly reliant on certain features of the input data. We show in Section 3.6.1 how networks trained with regularization are much more robust to the error introduced by our temporal networks, particularly when combined with our improved approximation. While this must be applied at training time, it requires minimal modification to the training stack and most networks already include some form of regularization.

## 3.3 A Temporal/Digital Systolic Architecture

While the proposed techniques shown in Section 3.2 demonstrate how artificial neural networks can be mapped to a temporal equivalent, prior implementations of temporal logic have either been fixed function [48, 2] or application specific [3]. This represents a significant gap between our proposed delay space networks and both ANNs and SNNs, as the ability to program new weights and biases is at the heart of these accelerators. To bridge this gap, we propose a novel hybrid temporal/digital systolic array to support the general class of delay space networks shown in Figure 3.5. Due to the algorithmic equivalence to ANNs, this systolic array leverages many of the same patterns as digital systolic arrays, exploiting reuse wherever possible and only falling back to digital when necessary. However, the transient nature of temporal computation makes many digital approaches to achieving these reuse patterns ineffective, as they require numerous domain conversions, which counteract the benefits of delay-space computation.

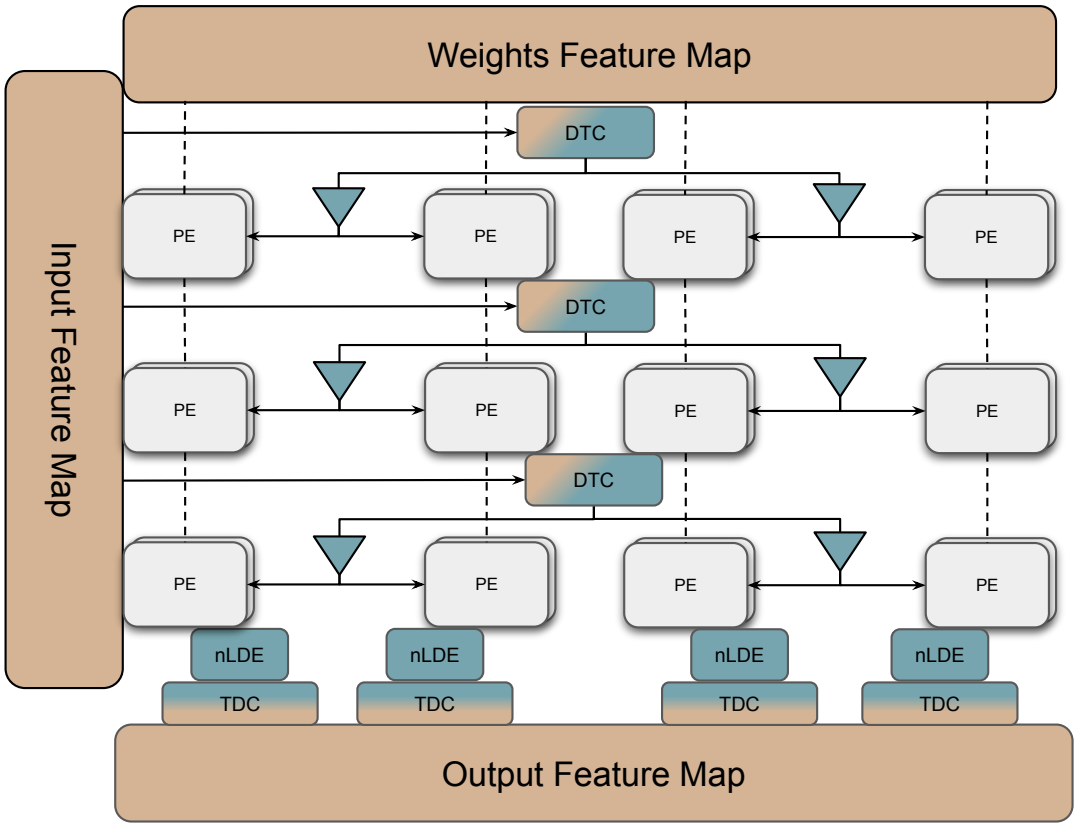



Figure 3.5: Architecture of the hybrid temporal/digital systolic array where all memory is digital, so accesses to the input and output feature maps require temporal conversions, either temporal to digital (TDC) or digital to temporal (DTC). The DTC is placed in the middle of the array to balance the broadcast path to each PE.

### 3.3.1 General and Programmable Operation

Programmable neural network accelerators begin with the ability to program weights, typically from memory into a register, which can be directly manipulated by logic. However, our transformed weights are encoded as delays, necessitating a circuit or device that has a variable propagation speed based on some external, programmable input. Several programmable delays have been proposed, including starved inverters [55], memristors [47], and pull-down networks [56]. These generally work by artificially increasing the resistance and capacitance within a single device or gate, attempting to create a linear relationship between the programming signal and the corresponding delay. Starved inverters and memristors can achieve this linearity, but are controlled through analog signals: steady state voltages and voltage pulses respectively, which would require another domain conversion step. Pull-down networks are digitally programmable, but offer a narrow range and high delay variability [57].

As an alternative, we use a cascading inverter approach [58], where each bit programs a mux that selects between a delay path and an immediate path, as shown in Figure 3.6. Each mux controls a delay path that is twice as long as the previous stage, providing $2^n$ delay paths. This leverages the fact that delay circuits use the same binary wires as digital logic, allowing digital control to direct and route temporal paths. However, this does limit the possible number of delay paths, forcing a fixed-point quantization.

Quantization, a technique that reduces the bit-precision of operations [59], is quintessential to most neural network accelerators, leveraging ANN's tolerance to error. However, due to the low dynamic range of these quantized values, it's often necessary to use scaling values to maintain network accuracy [60]. This introduces multiplication and rounding overheads that are typically a small fraction of the energy saved, but require additional, higher precision hardware units. Delay space quantization must use a similar scaling

scheme, partially to ensure all temporal values are positive. However, since delay space networks use a log number system, the multiplicative scaling can be simplified to addition. Additionally, the log number system allows for a wider dynamic range than an equivalent number of bits in a linear system, and is thus a promising candidate for many neural network applications [61, 62]

While a programmable delay is sufficient to create a general purpose architecture, without leveraging reuse, both spatially and temporally, the memory system will dominate the energy consumption of the system. Typically, when inputs are read from SRAM, classical architectures exploit spatial reuse through a store-and-forward method [63], allowing values to be used by processing elements (PEs) across space and time. However, this technique relies on registers, a structure that doesn't have a clear temporal equivalent. As we show in Section 3.3.3, memristive temporal memories could potentially act as a register, but the control logic makes it inefficient to implement as standalone devices inside the systolic array. Alternatively, the temporal signal can be converted to digital, stored in a register, then replayed on the next cycle. However, this domain conversion, particularly TDCs, is both area and energy intensive, and should be minimized wherever possible. Instead, we broadcast the temporal signals to all the PEs within a row to achieve the same level of spatial reuse as the store-and-forward approach. This broadcast can be achieved by placing the DTC physically in the center of the array. From the DTC, a branching routing scheme can be used to ensure that each PE receives the signal at the same time, similar to clock network H-trees [64].

After each PE's outputs are generated, most systolic arrays exploit temporal reuse by storing and replaying the result across subsequent cycles. However, the output of our delay space multiply and accumulate (MAC) unit is inherently temporal and again, cannot be stored effectively as-is. It could be immediately converted to digital, similar to methods used by other hybrid domain accelerators [12], but this conversion is expensive

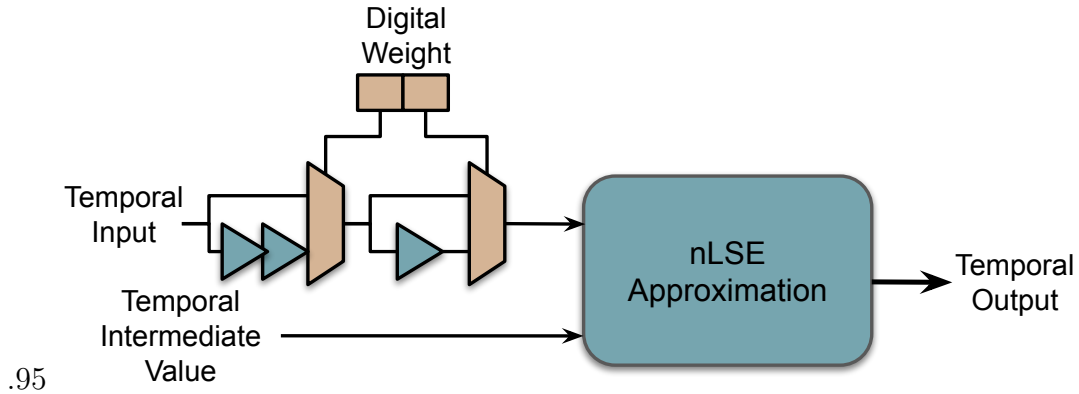
and doubles the energy consumed by each PE. Instead, we propose maintaining the temporal outputs by resynchronizing them with the next cycle's incoming data using temporal recurrence [48, 1].

This works by delaying the output of the signal so that it has the same scale as the next input, synchronizing the two signals. For instance, if two delay space values x' and y' arrive across two consecutive cycles, the actual arrival times are $t_x$ and $t_y + T$ where $T$ is the cycle time. However, addition in time is the same as importance space multiplication, so the $+T$ term acts as a scaling factor. To operate on the two values, the same scaling factor must be applied to both, which can be accomplished by delaying x' by the cycle time. Also, it's important that any operations performed this way must be distributive so that $f(x+T, y+T) = f(x,y) + T$, a requirement met by all delay space operations. While this does require delay to perform the synchronization, it can be combined with the delay inherent to the nLSE approximation. Since this approximation is along the critical path, minimal additional hardware is necessary to support recurrence.

This resynchronization along with temporal broadcasting exploits spatial and temporal reuse in systolic arrays, allowing the delay space spikes to leverage many of the same systems set up for ANNs. This reuse, alongside the digitally programmable delay elements, creates a general purpose architecture that can map all of the linear layers of a delay space network, and by extension all of the linear layers of an ANN.

### 3.3.2 Error Minimization through Temporal Logic Trees

In digital logic, floating point values suffer from an accumulation of errors as more computation is performed [65], and the shape of the computation has a large impact. A tree based reduction of computing floating point values has much less error than a pure linear reduction approach. This same phenomenon is compounded in the delay space




.95

Figure 3.6:

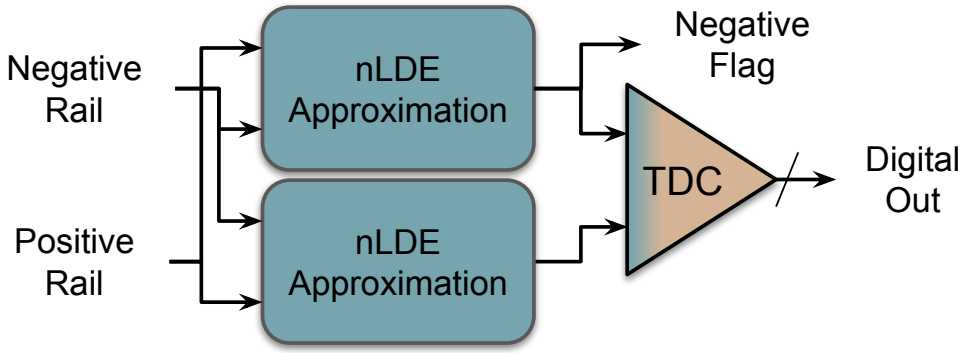



Figure 3.7:

Figure 3.8: a) The delay space MAC, equipped with a digitally programmable delay element. b) Paired nLDE-TDC readout with the automatically generated negative flag.

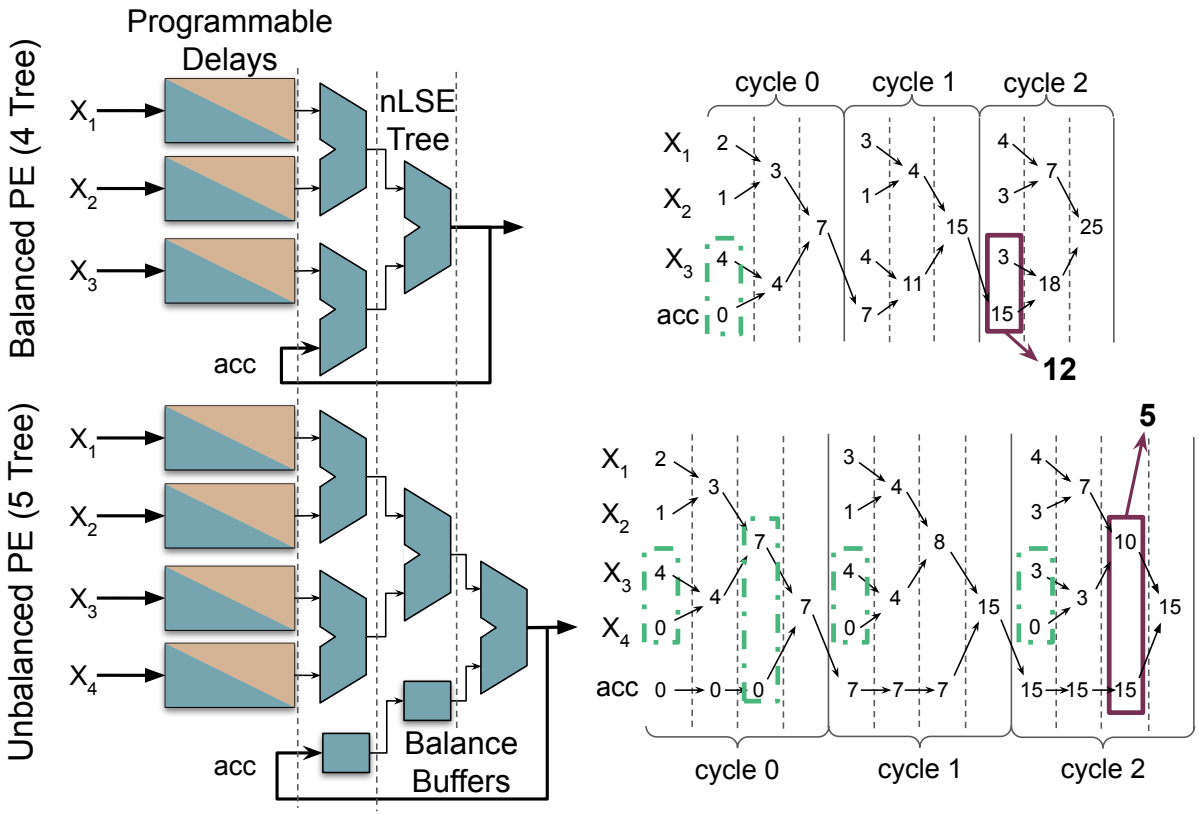



Figure 3.9: Two possible types of PE tree sizes, balanced and unbalanced (left) and their corresponding execution of an importance space set of data (right). We highlight the areas where the approximation is perfect with the dotted green boxes, and the maximum difference that needs to be approximated in the solid boxes. This max difference exemplifies how the PE structure impacts the nLSE input distribution that needs to be approximated.

approximations, where error minimization is critical due to the log-compressed nature of delay space. This is caused by two reasons, stemming from the approximation technique. First, nLSE with a larger input difference than the range of approximation becomes the min operation. This does not introduce much error in isolation, but a linear reduction creates a very small value in delay space. Since the reduction cannot include negative numbers, this value eventually becomes much smaller than the original input domain. At this point, every nLSE becomes min, losing all information that would be provided by additional accumulations. The second impact comes with the uneven accuracy of the approximation. Certain points along the approximation curve are closer to the true nLSE function than others, and so the accuracy is dependent on the absolute difference of the two inputs. This distribution of inputs changes as the reduction changes from linear to tree based, which we explore in Section 3.6.1.

In digital systolic arrays, the parallel nature of data representation constrains PEs to accept only one input, one filter, and one intermediate value per cycle [66, 67], resulting in a fully linear reduction. However, multi-input PEs have been shown in unary computing systems [11] and aggressively quantized systems [68] because both the required bandwidth and wire congestion is significantly reduced. This same approach can be used with our proposed array as shown in Figure 3.9.

Because the delay space nLSE approximations are designed for approximating a two-input function, a hardware tree must be used to fully reduce all of the incoming values. For maximum accuracy a tree large enough to accept all of the values in parallel would be used, but the large number of input values per neuron in neural networks makes this infeasible. Instead, the tree size is fixed and the values are folded across time, creating a semi-serialized reduction approach. If the reduction tree is fully balanced, then no extra hardware is required, but an unbalanced tree requires extra synchronization lines due to the constant offset introduced by each stage in the nLSE tree. Also, the PE tree size

creates a new trade-off between performance, energy, and accuracy. The larger the depth of the tree, the longer each cycle takes while requiring more memory accesses to ensure the PE is fully utilized. We explore these tradeoffs in Section 3.6.

### 3.3.3 Temporal Memory

While our proposed array up until this point shows how to minimize TDCs and DTCs, these conversions are still necessary to interact with memory, a classically digital structure. However, recently there have been large strides in creating memory systems that can capture timing of wave fronts, which can then be recalled on demand [47, 69, 70, 71]. These typically rely on *memristive*[72] devices, which have a programmable resistance based on the duration of a write pulse, leading to a direct relationship between the physical characteristic (resistance) and time.

Programming these devices requires a temporal signal and a reference signal, which can be directly generated by the clock. The memristor is connected in parallel with a transistor between the word line (the reference signal) and the bit line (the temporal signal). The transistor is then programmed by a select line that determines whether that specific device is being activated. When a device is selected for a write operation, the transistor is turned on, and when the reference signal goes high it creates a voltage difference across the memristor, which in turn causes the device's internal resistance to increase. Once the temporal signal arrives, the voltage difference becomes zero, preventing any more changes to the memristors state. As long as the difference between the reference and temporal signal is less than 40ns, which delay space operates within, this change in resistance is linear.

During a read the word line goes high at a lower voltage (to prevent modifying the memristor's state), which causes current to flow across the memristor, charging a

capacitor along the bitline. This capacitor can be connected to a gate that goes high once the voltage across the capacitor reaches a certain threshold. The time it takes to reach this threshold is linearly related to the resistance of the capacitor, causing the output pulse to be linearly related to the time between the reference pulse and the original programming signal. Using additional capacitors, this signal can create a one-to-one mapping of the programming and read signals. Although this operation introduces inherent delay, it is equivalent to a constant input reference shift.

This represents slightly more complex control circuitry than classical SRAMs, but its advantages far outweigh the cost. The input and output feature maps can be completely replaced with their temporal equivalent, totally removing the need for any TDCs or DTCs within the array. Additionally, a single device maintains a full value instead of 8 SRAM cells needed for every activation. This allows for a significant reduction in read/write parasitics, causing the memristive operations to dominate the energy consumption. While this is more expensive than a typical memory cell access, it still provides significant savings over it digital counterpart, as we analyze in section 3.6.2.

Precision of recall is the largest barrier to using this temporal memory. While this represents a challenge, memristive fabrication consistency has improved significantly, with device to device variability reaching less than 1% [73] and cycle to cycle variability small enough for 7 bits of precision [74] with current technologies. As memristor technology matures this variability will decrease, allowing for better distinction of stored wavefronts. While the variability will never completely go away, it will serve as a form of quantization, introducing error in a similar manner to the error introduced by temporal activations being quantized as they're stored digitally.

## 3.4 Dataflow for Temporal Operation

While Section 3.3.1 shows how reuse can be exploited at the micro-architectural level, it is well known that how this reuse is structured at the global level has a large impact on the energy of the full system for both ANNs and SNNs due to the orders of magnitude difference between the cost of a MAC and a memory access. ANNs must balance folding their inputs, weights and outputs across the systolic array and network layers, but SNNs typically have to balance all of these things across time as well [75]. This extra dimension increases the dataflow complexity and design space possibilities.

In general, because delay space performs the same operations as ANNs the array can leverage the simplicity of ANN dataflows. However, the spatial and temporal local reuse exploited by our architecture is based on the temporal values, which establishes limitations that prevent some dataflows. For example, dataflows that require inputs to be passed diagonally across the array are no longer effective because of the input broadcasting.

Also, many accelerators co-locate PEs with small scratchpads to maintain multiple sets of intermediate data to maximize utilization and reuse[76]. However, the recurrence that facilitates temporal reuse in our architecture incur increasing energy and noise as multiple cycle delays are required, making the flexible designs built on scratchpads impossible. Instead simpler data flows must be considered, like those used in systolic arrays that only allow for neighbor-to-neighbor communication. These data flows fall into three categories: weight stationary (WS), output stationary (OS), and input stationary (IS), and have been shown to be very effective [67].

IS requires that after each input arrives at a PE it is stored and re-used every cycle in the same PE. Again, the temporal inputs to our systolic array make this challenging. The input could be delayed to re-synchronize with the next cycle to maintain the

temporal representation as discussed in Section 3.3. However, the value that must be delayed is unaffected by the computation, so none of the recurrence delay is hidden by the computational delay. Instead there must be a long delay line to offset the signal by an entire cycle, consuming additional area and energy. Because of the incompatibility with the hybrid array we ignore IS and instead discuss how both WS and OS can be implemented using the array.

Implementing a WS dataflow is straightforward because the weights in the systolic array are fully digital. The programmable delay of each PE is latched to the incoming weight value and does not change until that weight has interacted with all necessary inputs. The temporal output of each PE is routed along the same column to the next PE where it is delayed to be synchronized with the input of the next cycle. This output is not being passed back into the same PE, no longer implementing recurrence. Instead, the same synchronization technique implements a temporal pipeline.

The OS approach only needs one PE to compute a single output pixel/value, while the necessary weights and inputs are streamed through the PE. The temporal result of each PE is synchronized with the next set of inputs, then fed back into the same PE that it came from, implementing recurrence exactly. Once the entire output is calculated, the temporal result is sent to be stored digitally. This output has been fully computed so only the activation function needs to be computed. For ReLU this means that half of the normalization nLDE circuits shown in Figure 3.7 can be removed, as only the positive rail is taken into account.

When considering a design that leverages temporal memories, the input and output feature maps are smaller and more energy efficient, while the weight feature map is still implemented digitally. Since this consumes much more energy per access compared to the other memories it's wise to try and minimize this access. Therefore, for the rest of this paper we leverage a WS dataflow to fully take advantage of the temporal memories.

While this approach more frequently leverages the nLDEs at the interface to the memory, it's more than offset by the energy efficiency gained by the cheaper access. Additionally, this periodic partial sum normalization can prevent recurring values from growing too large (fast) compared to the input signals.

## 3.5 Evaluation Framework

We used three different frameworks to evaluate our proposed system. First, to evaluate the accuracy of delay space neural networks we created a simulation framework based on TensorFlow[77]. This approach takes a pre-trained neural network and dataset, then converts all of the weights and inputs into delay space values. Next, it takes each layer and modifies the layer so that all calculations performed are replaced with their delay space equivalent (e.g., multiplication becomes addition and addition becomes our nLSE approximation). We add an option to perform these operations with noise based on prior analysis [78]. Generally noise comes from three sources: process, voltage and temperature (PVT). Process variation is correlated across large parts of the chip, which acts as a constant shift to all temporal operations. Since our operations are all shift invariant, this variation has minimal impact on the result of the computation [58]. For temperature, or random jitter (RJ) we model delay elements with independent Gaussian noise, so the noise grows at a reduced rate as the number of delay elements increase. Our voltage variation model assumes a distribution where the limits are determined by the potential swing in supply voltage. This model assumes a worst case scenario where voltage variation is correlated within a delay line, but uncorrelated across both space and time, so every operation is exposed to potentially maximum noise. Power supply jitter would normally impact the array globally, exposing all circuits within a cycle to the same shift, causing impacts to only appear over time. However, this models a rapidly

changing power supply to provide an extreme limit on power supply impact.

Next, to evaluate the energy efficiency, performance, and area of the delay space computations we use a 32nm technology model [32] to determine the energy consumption of the delay elements and basic operations in SPICE, using a 700mV supply voltage. We then use an analytical model to extend these values to determine the energy consumption of the delay space operations for a given unit scale, which determines how theoretical delay space values get mapped to physical timings (the unit scale is the physical delay that is interpreted as the value 1). For the TDC and DTC we use our SPICE modeling values and values from a simulated Vernier delay line [79]. To evaluate the energy efficiency of the full system we modify SCALE-Sim [80, 63], a systolic array architectural analysis tool. SCALE-Sim was modified to support the broadcast inputs of our proposed system and the multi-input PEs. This tool provides memory usage statistics, which we use in conjunction with Cacti 7.0 [81] to model the energy and area consumption of the digital memory systems. To evaluate our array in conjunction with temporal memory we rely on a 32nm RRAM process in NVSim [82], including the additional overheads of temporal reads and writes [47]. For the temporal memories analysis we include the TDC and DTC cost of all off-chip accesses that must interact with the temporal memories.

To evaluate our design we use the MLPerf Tiny benchmark suite[83], which provides neural networks for edge applications where ultra-low energy inference is necessary. In order to evaluate the impact of quantization on network accuracy we use a scale quantization method [84, 60] with no fine-tuning for the baseline importance space model, and use a transformed version of the same scheme for the delay space quantization. For a baseline and comparison we consider a digital systolic array and SATA, a state of the art spiking neural network accelerator[75]. The digital design is evaluated with computing units synthesized in Synopsys Design Compiler at 400MHz using a 45nm tech node [11], and SATA-sim to evaluate the SATA architecture. For both baselines we use tech node

scaling [85] to scale both results to 32nm to ensure a fair comparison.

## 3.6 Results

### 3.6.1 Hardware Impact on Accuracy

The configuration of the hardware in the delay space framework has a significant impact on the accuracy of the transformed neural network, so it is important to understand how the different hardware parameters can impact the performance of the neural network.

The hardware parameter that has the largest impact on neural network accuracy is the number of approximation terms, and in Figure 3.10 we show how the number of max-terms for nLSE approximation impacts the accuracy of the MLPerf Tiny benchmarks. For each of these graphs we fix the number of inhibit-terms for the nLDE approximation to 25, which we've experimentally determined to give sufficient network accuracy. The $x$-axis shows the number of approximation terms, while the $y$-axis shows the performance metric. We show two networks for each application, one trained with dropout for regularization and one trained without dropout. We also compare against our proposed approximation and prior approximations. For comparison, we count our proposed approximation to have the same number of terms as two times the original approximation (e.g a $2\times7$ term approximation appears on the graph as 14 approximation terms). In addition, each graph has two lines that indicate the original network's accuracy and the performance target set by the MLPerf Tiny benchmark.

In the original model, trained without dropout, our proposed approximation method has better accuracy for any given number of terms, with the exception of low-term approximations of anomaly detection. For the visual wake word and keyword spotting

benchmarks, a 2×3 term approximation is sufficient to reach the performance target, while the original approximation approach requires 15 terms to achieve the same target. This improvement is exemplified even further when applied to the regularized models. For the two networks already robust to noise, the original accuracy can be almost fully recovered with 2×5 terms, and 2×7 terms is sufficient to hit the performance target for every benchmark. While the original approximations see similar improvements from the use of regularization, it still takes 25 terms to reach the image recognition performance target.

However, this analysis assumes these approximations can be implemented perfectly in hardware, with full precision. Both quantization and hardware based noise will impact the final accuracy of the network, and need to be considered. To examine this impact we apply the noise models discussed in Section 3.5 to an 8-bit ResNet8 (the MLPerf Tiny image recognition network) trained with dropout. The results of the noise analysis on a 2×12 and 25 nLSE approximation term system are shown in Figure 3.11.

First we examine the impact of random jitter (RJ) on accuracy without voltage variation, shown by the blue curve and $x$-axis, as RJ is largely based on the device physics and can only be controlled by increasing the unit scale, which in turn increases the computation time. At large unit scales the accuracy degradation is minimal, and similar regardless of which approximation method is used. However, as the unit scale reaches 1ns and shorter, our proposed approximation accuracy decreases less than the original, staying above the performance target with a 750ps unit scale.

We then perform an experiment that considers both RJ and voltage variation, assuming a base 1ns unit scale for RJ. Voltage variation is swept, parameterized by the potential voltage swing as a percentage of the original supply, and shown in the red lines of Figure 3.11. This demonstrates how voltage based noise can have a much larger impact on network accuracy. However, as discussed in Section 3.5, this is assumes a system

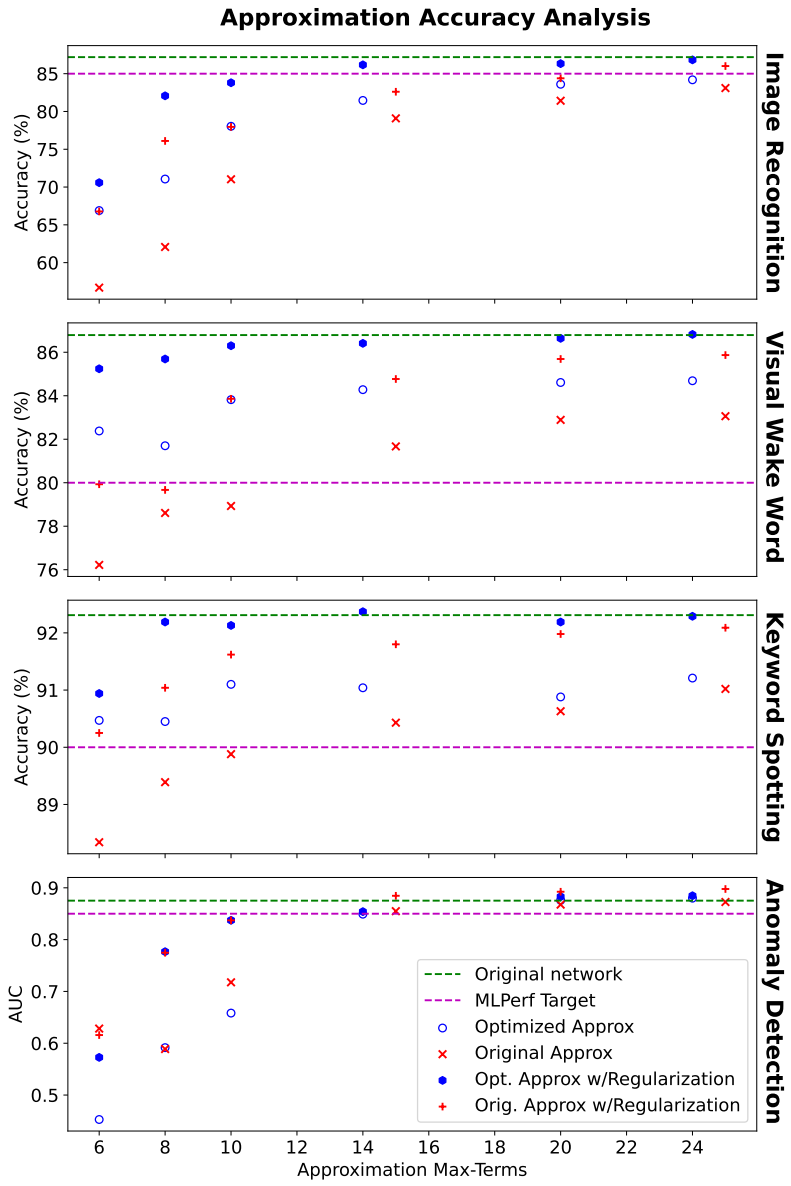



Figure 3.10: Network accuracy impact of the original delay space approximations (Single Approximation) and our optimized approximations (Dual Approximations) (10 max-terms for the original approximation correspond with 2×5 terms for our proposed optimized approximation). The number of inhibit-terms for delay space subtraction is kept constant at 25 terms for each network. We show the converted network accuracy both when the original network is trained with and without regularization. The top horizontal dashed line represents the original ANN accuracy, while the purple line represents the minimum performance required by the MLPerf guidelines.

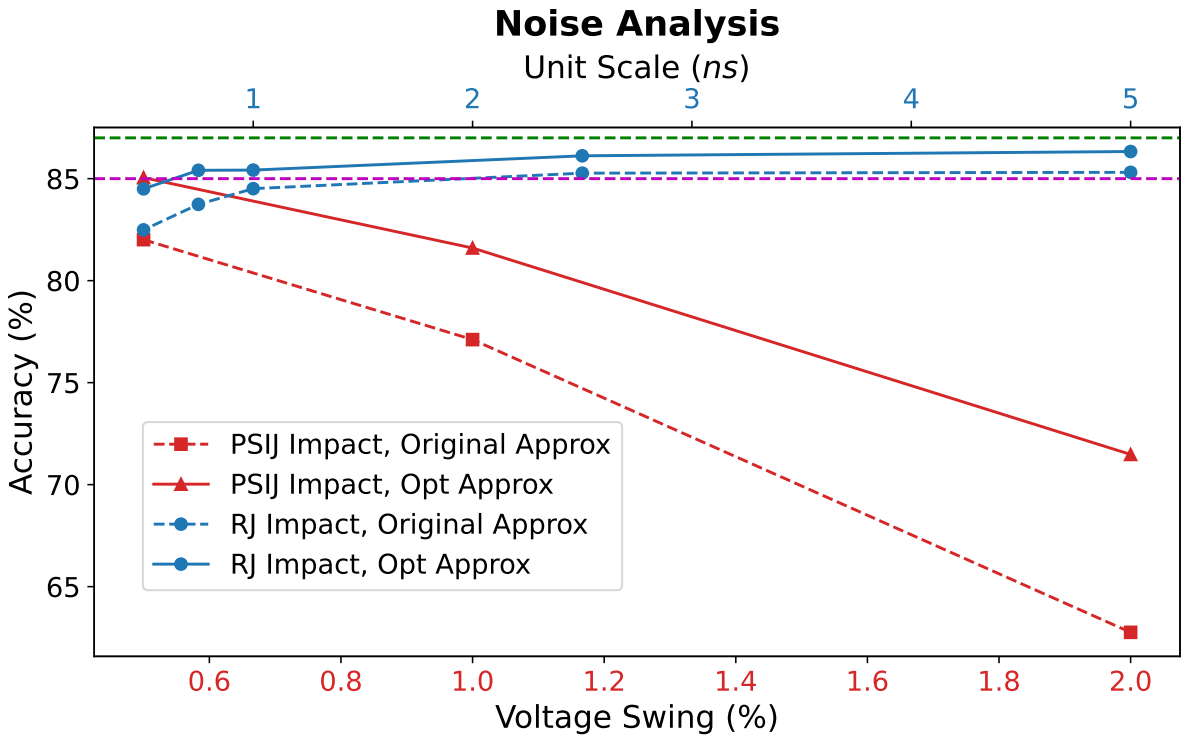



Figure 3.11: Analysis of how random jitter (RJ) and voltage variations impact the accuracy of delay space implementation of an 8-bit ResNet8 for image classification. The RJ analysis is performed without voltage variation and is shown in blue, while the voltage variation analysis includes RJ with a 1ns unit scale.

much noisier than any realistic hardware, with potentially maximal variation occurring multiple times every cycle. The primary takeaway is our improved approximation is less impacted by noise, even under extreme conditions.

These two experiments show at a circuit and logic level how the approximations can impact neural networks, but architectural design decisions also impact the overall accuracy, as discussed in Section 3.3.2. In Table 3.2 we show how 4 different PE tree configurations can impact accuracy ($n$-tree indicates a PE tree with $n$ inputs), assuming a 2×12 nLSE approximation terms. We consider 3 balanced tree configurations, and one unbalanced configuration (9-tree).

While most of the benchmarks have minor accuracy fluctuations with changing PE

Table 3.2: MLPerf Tiny benchmark accuracy as the PE tree size is swept, including one unbalanced tree configuration.

| | 9 Tree | 8 Tree | 4 Tree | 2 Tree |
|---|---|---|---|---|
| Image Recognition | 86.36% | 85.72% | 86.41% | 85.65% |
| Visual Wake Word | 86.57% | 86.42% | 86.65% | 86.44% |
| Keyword Spotting | 92.04% | 92.72% | 91.35% | 91.21% |
| Anomaly Detection | .857 | .48 | .47 | .46 |

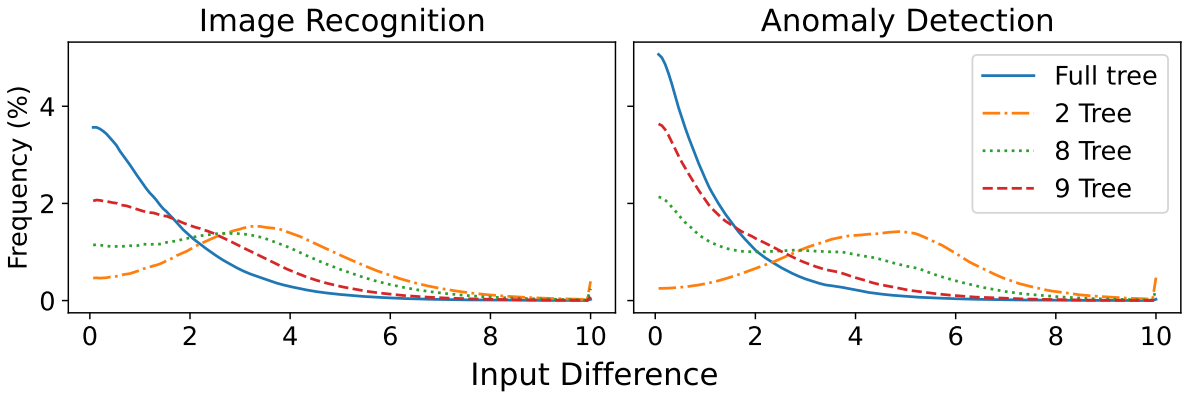



Figure 3.12: Histograms of the nLSE approximation input values for the image recognition and anomaly detection benchmarks as the PE tree size is changed.

tree sizes, anomaly detection degrades drastically. While the performance can be recovered with a large, unbalanced tree, balanced PEs have much worse accuracy. The reason for this degradation is shown in Figure 3.12, displaying how the approximation input distribution changes drastically with the tree size. When anomaly detection uses balanced tree PEs, it creates a much larger spread of possible input values than image recognition, exemplified by the differences in the 2 tree PE histograms and the wide flat region that occurs in the 8 tree. The approximation is the most accurate as values are close to one another, and less accurate as they move further apart, so this larger spread means more operations are being approximated poorly. Using a 9 tree PE anomaly detection moves the distribution significantly closer together, where the approximation is better, allowing it to recover the accuracy of a full-tree implementation.

It may be possible to tune the placement of the approximation terms to be better suited for the larger range present in anomaly detection, but this would result in worse performance for the other benchmarks. In the pursuit of a generally applicable delay space systolic array we optimize for the common case. However, this does mean some applications and networks may not be suitable for this approach. This histogram analysis can be easily used to determine if a new network will perform well on a given accelerator architecture.

### 3.6.2 Architectural Analysis

Using the evaluation methodology discussed in Section 3.5 we perform a design space exploration for two of the MLPerf Tiny benchmarks for both digital and temporal memory systems, shown in Figure 3.13. Our design space exploration varies the height and width of the systolic array, along with the size of each processing element. We fix the input buffer, output buffer and filter map to 32KB, and use 2×12 and 25 approximation terms

for nLSE and nLDE respectively with a 1ns temporal mapping. Each network is quantized to 8 bits and use a WS dataflow. We show the area of the systolic array on the $y$-axis, while the $x$-axis shows the energy delay product (EDP) of the systolic array and on-chip memory, excluding leakage power. For this design space exploration we only consider architectures that have a smaller physical footprint than an 8 bit 12×14 digital systolic array, showing how a single delay space MAC unit is significantly smaller than a digital multiplier.

While the fundamental MAC unit is small, the nLDE and TDC are quite large, limiting the maximum possible width for designs that meet the area constraint when using the digital memory system. Because of this limit, the exploration reveals that for edge inference tasks, narrow (smaller broadcasts) and tall (more IF map bandwidth) designs tend to provide better EDP with a smaller footprint, with most Pareto optimal designs having a width between 7 and 9 PEs. Conversely, the temporal memory system can have wider arrays because the expensive TDCs are no longer necessary, creating many 2-tree designs along the Pareto optimal frontier. Additionally, the visual wake word application benefits greatly from 2-tree designs, while for all other benchmarks and designs the 4-tree designs perform better.

Using this analysis we select 3 designs that are generally along the Pareto optimal frontier. We select a 9×8 array with a 4-tree PE from both designs to highlight the benefits of temporal memory, referred to as DS DM (digital memory) and DS TM (temporal memory) for comparison against other systolic arrays. We also select a temporal memory design that is less than half the size of a digital array as another comparison point, referred to as DS TM Tiny.

We then compare these designs to an 8-bit 12×14 digital systolic array, a common edge inference configuration [66], and SATA, a SNN accelerator. We evaluate SATA using SATA-sim, and assume each benchmark is implemented with 20 time steps, a conservative

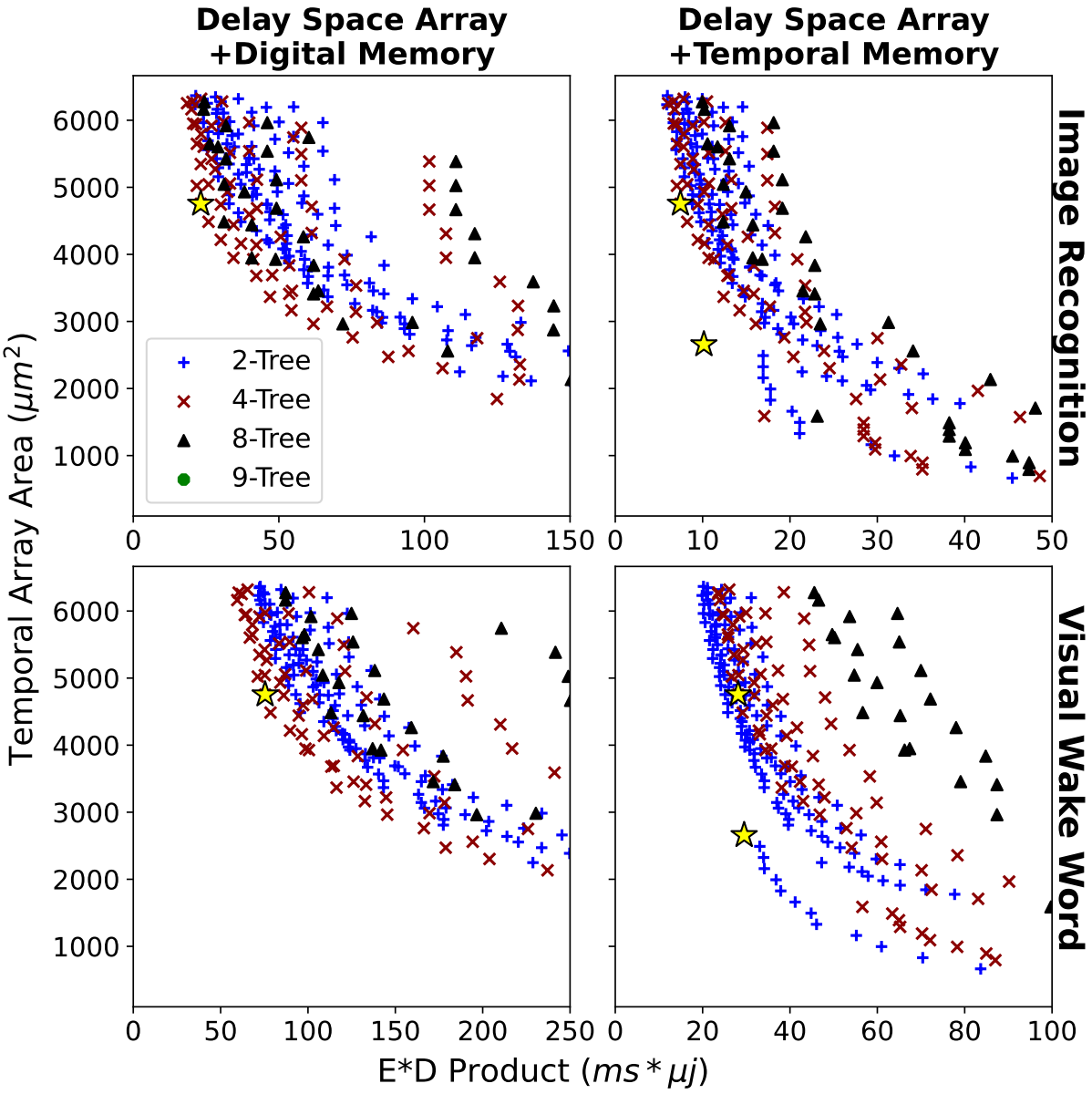



Figure 3.13: Design space exploration of potential temporal array sizes, comparing the overall area to the EDP. 3 designs are highlighted across the benchmarks with a star corresponding to DS DM, DS TM and DS TM Tiny.

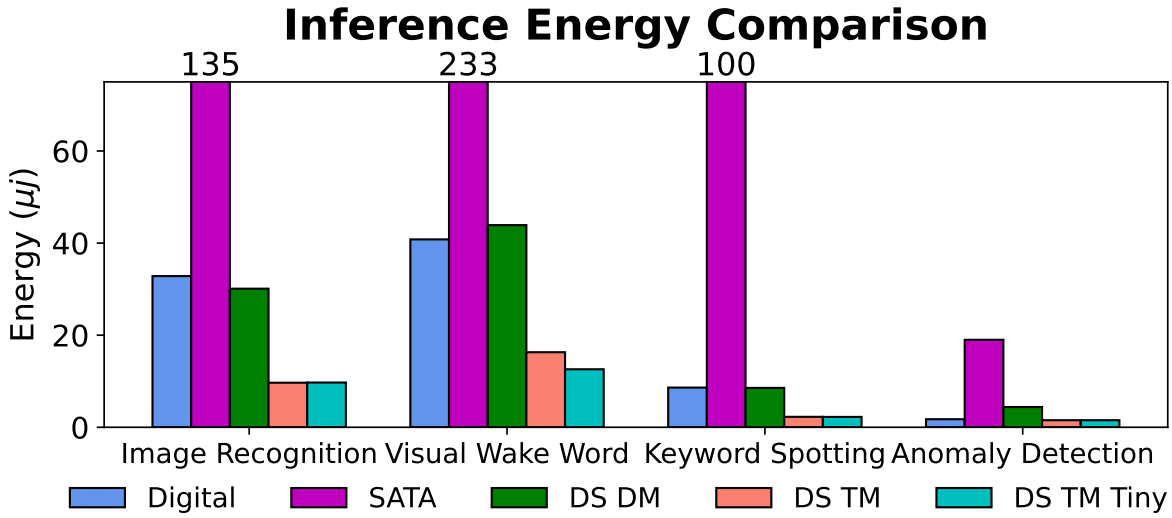



Figure 3.14: Active Energy comparison between our proposed delay space systolic array, an 8-bit digital implementation and an SNN systolic array, SATA [75].

| Arch. | Image Recognition | Mobile-net | Keyword Spotting | Anomaly Detection |
|---|---|---|---|---|
| DS DM | 1.04 | 2.33 | 0.49 | 0.16 |
| DS TM Tiny | 1.87 | 3.32 | 0.68 | 0.34 |
| SATA | 5.14 | 8.5 | 3.5 | 0.66 |
| Digital | 0.26 | 0.45 | 0.09 | 0.05 |

Table 3.3: Inference latency (in $ms$) of the delay space systolic array compared to the SATA SNN architecture and a digital systolic array.

estimate as most direct ANN-to-SNN networks use 32 to 1000 time steps [45]. For both designs we assume the same DRAM configuration as our design space exploration, then show the on-chip inference energy consumption of the architectures for each benchmark in Figure 3.14. We show the latency for a single inference in Table 3.3, excluding DS TM since it has roughly the same performance as DS DM.

Compared to SATA, our hybrid systolic array performs much better, even when using digital memories. DS DM achieves an 11× improvement in energy efficiency over SATA for keyword spotting, while performing 4.5× better for image recognition. This also comes with significant speedups over the SNN approach, saving over 0.5 to 6ms across the benchmark tasks. These results show the currently attainable benefits of our unified spiking ANN approach, leveraging classical dataflows for significant energy efficiency over methods leveraging temporal dynamics. While it has been shown that SNNs can leverage fewer time steps when utilizing advanced retraining schemes, DS DM still provides at least 2× energy improvement for each benchmark over a 6-step network on SATA while remaining strictly faster. However, when compared to a digital accelerator this digital memory approach consumes roughly the same amount of energy as a digital array, with slight variations across the benchmarks. While in practice digital arrays need to slow significantly to achieve this energy efficiency, the EDP difference is significant.

This stems in part due to the cost of extra domain conversions, and in part due to the memory systems dominating the cost of inference. This illustrates the need for temporal memory in order to drive continued improvements in energy efficiency. Using this design our systolic array becomes significantly more energy efficient, even when using an incredibly small design. DS TM and DS TM Tiny provide 2.5× to 3.5× improvement in energy consumption compared to the digital design for all benchmarks except for anomaly detection, while achieving 12× to 45× better energy efficiency compared to SATA. This shows the potential future for our proposed array, demonstrating how operations within

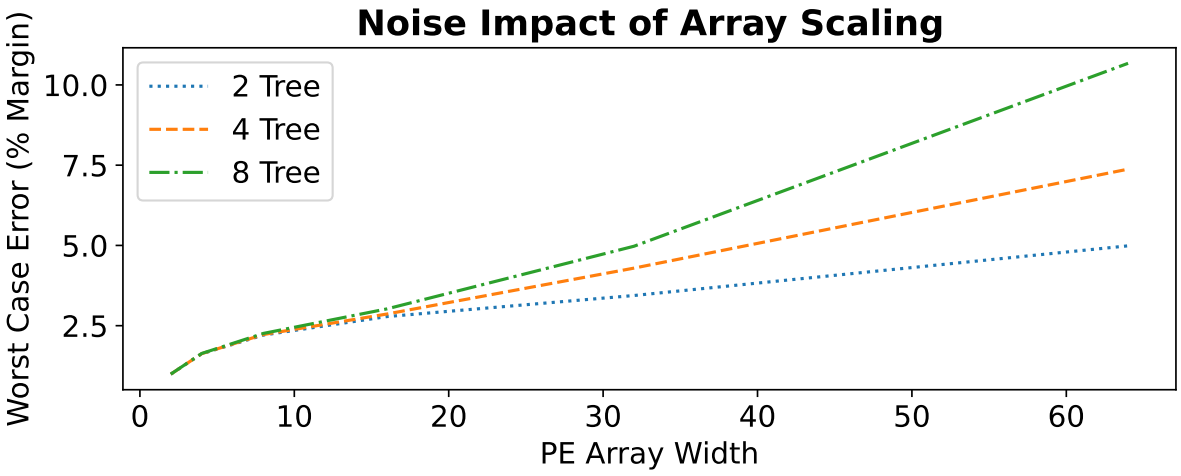



Figure 3.15: Analysis of broadcast noise as the width of the array scales.

a fully temporal system can provide significantly better energy efficiency than even the most aggressive digital approaches.

### 3.6.3 Implications of Broadcast Scaling

While we show several design points that provide very energy efficient inference for the MLPerf Tiny benchmarks while maintaining accuracy, the final question is how this technique scales to larger systolic arrays. In terms of accuracy, larger networks tend to be more resilient to error, especially when trained with regularization techniques. This can be seen in Figure 3.10 and Table 3.2, where the visual wake word application is much less affected by both the number of terms and the PE tree size. This application leverages the Mobilenet [86] network architecture, which is by far the largest network of the benchmark suite, showing how size can lead to better noise resilience.

For larger networks, a larger array may be necessary to meet performance goals. This can generally be achieved in two ways: increasing the size of the array (scale up), or adding more arrays in a tile-wise manner (scale out). The trade-offs between the two have been well studied [80], and the delay space systolic array shares many of the

same trade-offs because it leverages the same dataflows as ANNs. The only additional complexity is the broadcast operation, which introduces increasingly more noise as the array grows.

We analyze this impact in Figure 3.15, where we assume a system that augments the broadcast operation with buffers at every split of the broadcast tree, and a buffer every $50\mu m$ to ensure sharp rising/falling edges. We calculate the maximum error from this system assuming a maximum voltage variation of 5 mV in a system with a 700 mV supply, and three standard deviations of RJ. This maximum timing error is then divided by the timing margin that would cause a TDC at the PE to flip a bit.

Larger PE trees introduce noise at a faster rate as the array width scales because each PE has a physically larger footprint. This causes the required broadcast distance to increase, allowing for larger worst case error. However, even the larger designs stay well under the timing margin, even assuming worst case noise. This small amount of noise, coupled with noise-robust neural networks, allows the array to scale to several times the size of the arrays we've analyzed.

Other work has instead focused on purely temporal computation, using wavefront computing to perform energy efficient DNA sequencing [87, 88, 89] and graph operations [2]. Boosted race trees [3] even leverage the basic race logic framework for energy efficient decision tree inference. However, race trees have been shown to have limited effectiveness when applied to larger inference tasks [3]. Compac [90] uses an analog temporal memory to perform addition and multiplication, thus allowing it to perform inference. While this achieves energy efficiency it comes at the cost of performance since the operations become linear, forcing iterative multiplications.

## 3.7 Conclusion

We present a new approach to machine learning inference which bridges the gap between spiking neural networks (SNNs) and traditional artificial neural networks (ANNs) through temporal arithmetic. We show how temporal computation can be extended to implement a log-transformed version of neural networks by interpreting neural weights as delay elements and activations as a single "spike" in time. This transformation simplifies the dot-product calculation at the core of the network, and uses carefully optimized approximations for the simplified computations.

Implementing full neural networks directly in this log-time domain introduces significant new challenges and opportunities. First, network weights must be easily loaded and unloaded while also influencing the delay of signals in the system to achieve the required scaling. We develop a hybrid temporal/digital approach that enables weights to be migrated fully digitally while allowing delay operations to remain entirely in the temporal domain. Further improvements can be achieved by integrating previously demonstrated temporal memory systems — such systems have never been evaluated in the context of an end-to-end application. Additionally, we show that temporal arithmetic presents unexpected opportunities for energy savings in neural network systems. Two of the most common additional operators, max-pooling and ReLU, are not only exceedingly low-cost (e.g., max can be implemented with a single logic gate) but also simplify temporal arithmetic by eliminating much of its complexity — for example, ReLU removes the need for negative rails and their interactions with both positive and negative rails.

Through a careful analysis of these systems from low-level circuit noise up to application-level evaluation, we show how such a scheme is not only viable, but that the different hardware parameters can impact the accuracy of the converted neural network, creating a new set of energy and accuracy trade offs. By effectively marrying the simplicity and

energy efficiency of SNNs with the well-established structure and functionality of ANNs, this work opens new avenues for research and development in low power embedded machine learning systems.

# Chapter 4

# Architecture for Energy Efficient ZO Training

## 4.1 Introduction

One limitation of the temporal neural networks proposed in the previous chapters is their incomparability with the dominant training strategy, backpropagation. The approximation functions rely on non-differentiable functions, so gradient descent cannot be directly applied to optimize the delay-based weights. Instead, alternative optimization techniques must be explored, both to determine better weight values and to determine better approximation constants. Therefore, in this chapter explores an emerging optimization strategy, zeroth-order (ZO) optimization. However, during this exploration it became clear that this stochastic-based optimization approach can achieve significant benefits from new digital architectures. Therefore, This chapter primarily explores how classical architecture techniques can be used to create a platform for energy-efficient finetuning.

### 4.1.1 Motivation and Contributions

Increasing amounts of computation are being pushed towards resource limited devices: onto phones, wearable devices, environmental monitoring systems, and beyond. This demand for increased compute is largely driven by the rise of neural networks; smart, data-driven models tailored to specific applications provide immense utility across a variety of fields. However sensors can age [91] and environments can change [92], requiring machine learning engineers to design models that are generalizable across all potential deployment scenarios. This one size fits all approach can be incredibly challenging [93]. One approach to combat this problem, especially when deploying transformer models, is to use a larger network. For example, the smallest Qwen2.5 model [94] has 0.5 billion parameters and has an accuracy of 52.4% on the Stanford Sentiment Treebank (SST2 [95]), while the 7 billion parameter model has an accuracy of 63.6%. Unfortunately, these larger models quickly become too large to fit on most resource constrained devices. The alternative is detecting when the model is operating on out-of-distribution data [96, 97], then finetuning the model to its specific environment. For instance, the 0.5B Qwen2.5 model can be finetuned to have an accuracy of more than 90%, well outperforming the non-finetuned larger model.

However, on-device finetuning faces significant challenges. Traditionally finetuning has been performed using backpropagation [98], which can introduce prohibitive memory and compute costs [99, 100]. These overheads are amplified in modern transformers, where each additional token requires more memory to store intermediate activations, and introduces additional compute on the backwards pass. Additionally, the optimizer state, which is crucial for finetuning transformer models, requires two to three times more memory than the model itself [101]. Recently, however, alternatives to backpropagation-based learning have been proposed [102, 103, 104] which instead rely on multiple forward

passes to perform learning. Many of these approaches rely on zeroth-order (ZO) gradient estimations, which use the difference between two forward passes to determine parameter updates [105].

The difference between the forward passes is driven by a carefully controlled noise being added to every parameter in the network. If one of the forward passes performs better than the other it indicates that the parameters should move in the direction of that pass' noise. This approach has shown to be remarkably effective on fine-tuning modern transformer architectures, achieving similar results as those achieved by backpropagation [106, 107]. Unfortunately, the speed of convergence is significantly slower, requiring 100 times the number of training steps to achieve these results. These extra training steps are particularly bad when considering edge deployment, as it would require capturing and saving model inputs for several steps. This data capture and replay can quickly become unwieldy and expensive, making the memory demand closer to that of backpropagation.

In this chapter we show how this convergence can be significantly sped up using careful architectural co-optimization. Convergence speed is inversely proportional to the size of the model, or the number of dimensions that need to be perturbed [108], but prior approaches to reducing the dimensionality introduce excessive extra computation [109]. This dimensionality reduction works by noising each parameter individually, rather than all together, changing the paradigm from two forward passes to $d$, where $d$ is the number of parameters in the model. In this work we explore a middle ground, where groups of parameters are noised together, and design an execution graph which leverages the repeated use of parameter weights.

We further optimize this execution flow by demonstrating how each subsequent forward pass exhibits high data similarity to the first forward pass. Leveraging this observation we propose a novel difference-based execution scheme that allows for aggressive quantization and value reuse, significantly reducing bandwidth and compute overheads

introduced by the additional forward passes. We then demonstrate how both the quantization and de-quantization can be efficiently implemented to introduce minimal overheads to the underlying compute units.

While these contributions can significantly increase the efficiency of the forward component of ZO training, we note that the parameter update step can also consume a significant amount of energy. To limit this overhead while still maintaining as much finetuning flexibility as possible we develop a hardware unit that dynamically determines which weights are important for a given training step. This allows for careful, sparse noise injection, maintaining utility and flexibility while also significantly reducing the number of parameters updated every training step.

Specifically, our contributions are:

- We propose a novel, low-power ZO training accelerator designed to support our proposed interleaved execution dataflow for simultaneous ZO forward passes.
- We present a difference-based quantization scheme and corresponding hardware implementation, showing how quantization can improve training accuracy for severely energy-limited systems.
- We demonstrate a hardware-driven approach to ZO training dimensionality reduction, using top-K selection to dynamically reduce which parameters need to be updated.
- We analyze the effectiveness of our proposed hardware optimizations, analyzing the tradeoff between convergence speed and computational resources.

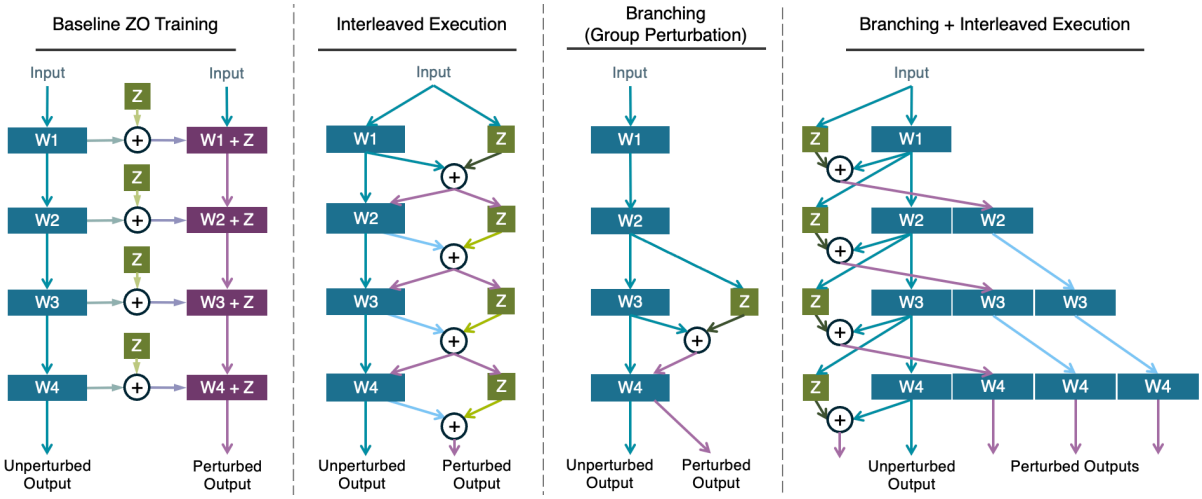



Figure 4.1: Execution graphs of the different approaches to the forward passes necessary for ZO training, where each edge is a set of activations, and each box represents a weight matrix. The far left is the current state of the art approach that relies on sequential operation. A full forward pass is computed, then each layer is perturbed, and then a second forward pass is computed. The second graph shows our proposed decomposition and parallelization strategy that allows for the two forward passes to be interleaved together. The third graph demonstrates our group perturbation policy generating a branch part way through the execution, allowing for better gradient estimates of the third layer. Finally the far right graph puts together our proposed interleaved execution and branching execution to maximize parallelization and parameter reuse while minimizing extra computation.

## 4.2 ZO Training and Parallelization Opportunities

Zeroth-order training operates by exclusively using forward passes, and injecting noise into each parameter between forward passes [106]. Each forward pass operates on the same inputs, with the parameter perturbations causing the model to perform slightly better or worse. An estimate of each parameter's gradient can be made using the difference between two forward passes and the exact noise that was applied to the parameter. Intuitively, this can be viewed as trying something multiple times, and adjusting your understanding of how to accomplish the task according to whichever attempt provided the best outcome. Formally it can be described as:

$$\widetilde{\nabla}\mathcal{L}(w) = \frac{1}{Q}\sum_{i}^{Q}\frac{\mathcal{L}(w+\epsilon z_i)-\mathcal{L}(w)}{\epsilon}$$

Where $w$ is the parameters in the model, $\epsilon$ is a small value to assist with smoothing, $Q$ is the number of queries (or forward passes), and $z$ is a randomly generated vector with length equal to the number of parameters in the model. This perturbation is added to every weight within the network in a given step, and then each weight is updated using the predicted gradient and the corresponding perturbation:

$$w = w - \widetilde{\nabla}\mathcal{L}(w)z\alpha$$

This update occurs according to ZO stochastic gradient descent (SGD) [110], where $\alpha$ is the learning rate. ZO training does not provide exact gradients, so it struggles to converge when performing training from scratch [111], but has achieved remarkable performance when used for finetuning tasks on transformer models [112, 106, 113], even outperforming traditional backpropagation and optimizers for certain tasks. However, this finetuning occurs much slower because in order to ensure a stable gradient the learning rate must be decreased according to the dimensionality (number of parameters) of the network [114]. In order to achieve results comparable to classical backprop, ZO training approaches

often use 80 to 100 times more training steps [106, 112].

These additional training steps present significant overhead, especially in edge scenarios where a device might not have the resources to capture and replay data 100s of times. To combat this overhead an emerging approach called coordinate-wise gradient estimation (CGE) has been proposed [115]. CGE operates by noising each parameter in its own forward pass, resulting in a better gradient estimation, reducing the required number of training steps. However, CGE requires $d$ forward passes per training step, where $d$ is the number of parameters in the model. Even with sparsity driven approaches [109] this drastic increase in forward passes is untenable. In an attempt to minimize the number of training steps without drastically increasing the amount of computation per step, we introduce a middle ground between CGE and classical ZO training.

We propose grouping parameters together where each group is perturbed in the same forward pass every training step. Grouping is done in a structural manner, with a minimal grouping being all of the parameters in a linear layer. Then, if the grouping size is larger than a single layer, sequential layers are grouped together. As opposed to other grouping approaches [116, 111], we do not freeze any groups or parameters, but generate as many forward passes as parameter groups. This creates a trade-off space, with smaller groupings needing more forward passes per step, but requiring fewer total training steps. In Section 4.7 we demonstrate this tradeoff, but note that without architectural support these additional forward passes quickly introduce excessive overhead.

To reduce these overheads we propose two innovations: interleaved execution and branching forward passes.

### 4.2.1 Interleaved Execution

Prior ZO training work assumes sequential execution of all forward passes as shown in the left side of Figure 4.1. First, the unperturbed forward pass is computed, then all of the parameters are updated to include noise, and then a second forward pass is computed with the noised parameters. Finally, after the gradient has been estimated, each parameter is relaxed to its initial value before being updated. The energy consumption of this training step is dominated by the number of DRAM accesses necessary, requiring each parameter to be loaded on chip four times. Parallelizing the forward passes and noise generation would be able to halve the number of times each parameter is accessed.

However, because the second forward pass operates on perturbed weights it is difficult to treat the two inferences as a single batched operation. To overcome this limitation we note that the addition of noise to the weights within a linear layer is fully separable. Matrix multiplication, while not commutative, is distributive, meaning that:

$$O = a \times (W + z) = a \times W + a \times z$$

Here $O$ is the output, $a$ is the activations, $W$ is the weight matrix and $z$ is the noise matrix. We discuss parallelization strategies for $a \times z$ in Section 4.3, but this enables fully batched operations of each linear layer as shown in the second graph of Figure 4.1. This batched operation, alongside a weight stationary dataflow, enables multiple ZO forward passes to be interleaved together, achieving much better throughput and energy efficiency when compared to the sequential approach taken by classical ZO training.

### 4.2.2 Branching execution

While interleaved execution can significantly reduce the overhead introduced by the additional forward passes, the middle ground between CGE and traditional ZO training

can be exploited for even further savings. The parameter grouping operation is shown as the third graph in Figure 4.1. In this figure only the third layer is being perturbed, which means that all previous layers only need to be executed once. Then, when the third layer is being executed it generates a "branch", where noise is added to the layer output and stored separately from the original forward pass (which we refer to as the "clean" branch). Only the layers after this third layer will need to be executed twice, once for the original forward pass, and once with the perturbation. This results in two outputs which we can use to update just the third layer, providing a much better gradient estimation compared to noising the entire network, while only adding computational overhead after it becomes necessary. However, to perform a full training step, every parameter group or layer in this example must be updated. This could be done by repeating the forward pass with a single branch for every layer, but this approach introduces sequential operations.

Instead, we combine this branching execution with our interleaved operation. For every parameter group that needs to be noised together (each layer in this example), a branch is created from the original, clean branch. This branch is then batched with all of the current branches, allowing each matrix multiplication to share the same parameters as shown on the right side of Figure 4.1. As the execution progresses the number of branches increases, so that at the end there is as many outputs as parameter groups in the network (5 outputs in this example). This allows us to produce many outputs, while only actually computing half of the computations required to perform an equal number of forward passes. We refer to this technique as branching, interleaved execution.

While interleaved and branching execution provide the potential for significant energy savings compared to a naive ZO implementation, both techniques rely on the ability to efficiently compute the perturbation impact alongside the interleaved original linear layer.

## 4.3 Parallel Processing Elements

We propose the processing element (PE) shown in Figure 4.2 to flexibly support the various scenarios that appear during ZO training. We assume this processing element operates within a TPU-like systolic array using a weight stationary dataflow. The goal of this processing element is to support the following different scenarios that appear within branched, interleaved execution:

- Creating a new branch from the clean forward pass
- Adding noise to an already generated perturbed branch
- Executing multiple perturbed branches in parallel

We describe each of these operations below, but note that this PE is designed to be able to quickly switch between each scenario without changing the dataflow or memory operations.

*Generating a new branch (AB = 10):* Naive approaches to performing the separable activation-noise matrix multiplication ($a \times z$) involve sequential computation of the original linear layer and the noise multiplication. First, matrix multiplication between the interleaved activations and the non-perturbed weights would be executed. Then the array would be reset, and just the activations from the clean branch would be used for a separate matrix multiplication with the generated noise matrix. Re-accessing these activations introduces significant overhead because many of the values will have already been evicted from the local SRAM. Additionally, these activations are not sequential in memory, instead being interleaved with the other branches.

To avoid this overhead, we introduce an execution scheme where the noise calculation occurs in parallel with the weight calculation. To facilitate the parallel computation we introduce a noise register, which accompanies every weight register. The activation, when

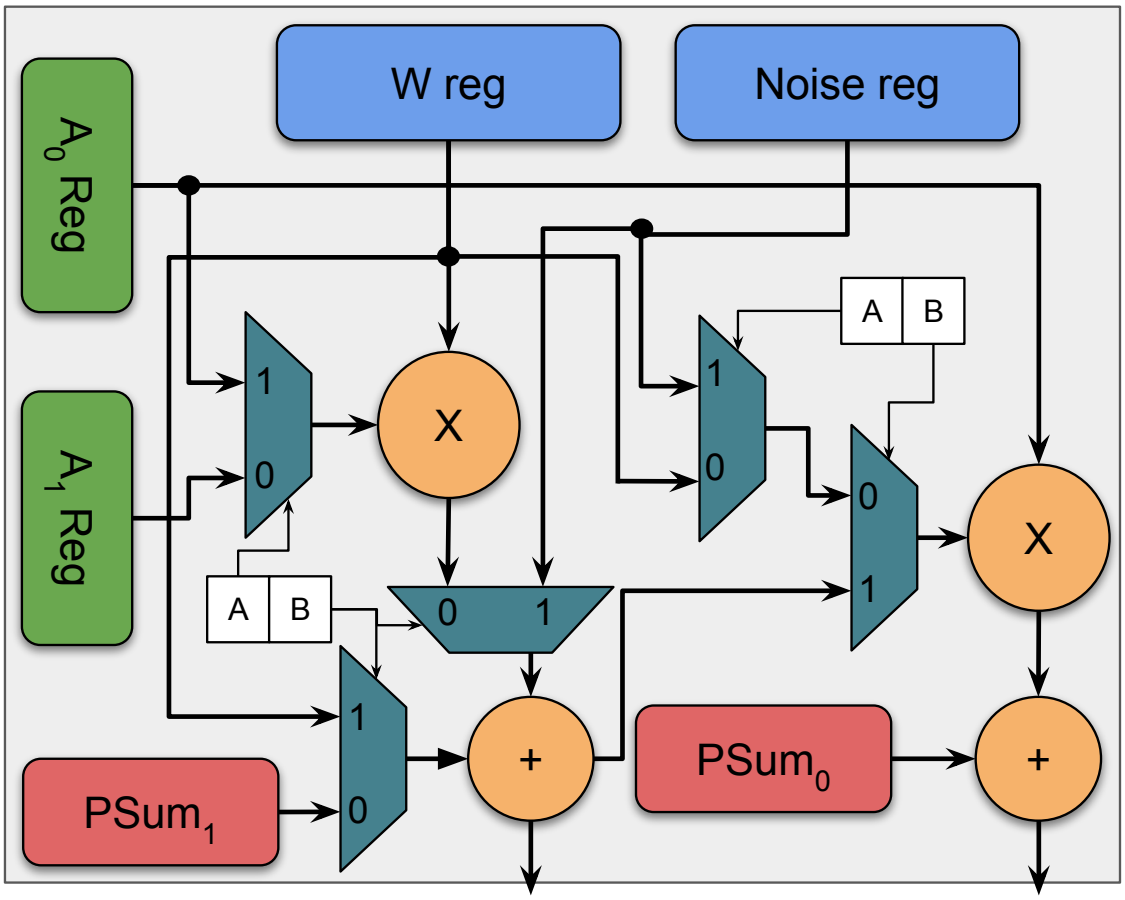



Figure 4.2: Our proposed processing element to automatically handle the three possible scenarios that arise when executing branching computations. The A and B registers (replicated for visual clarity) are 1 bit state registers that can be passed along side the data to configure the correct paths. AB=10 sets up the PE to generate a new branch, AB=01 will add noise to an existing branch, and AB=00 will perform parallel branch execution.

read from its corresponding register, is broadcast to two multipliers and accumulated into separate registers. One partial sum register stores the original, clean matrix multiplication, while the other stores just the impact of the noise. When the computation finishes, both matrix multiplication results are available and can be stored as-is (see Section 4.4).

*Adding noise to an existing branch (AB = 01):* When multiple layers are grouped together, it will be necessary to add perturbations to an existing branch. This perturbation does not need to be separate as in the first scenario because the unperturbed multiplication isn't necessary for further computation. However, storing the weights in a register with the noise already applied adds unnecessary overhead to interleaved execution. This is because the next activation to arrive at the processing element will not be a part of the same branch, so the weight register will need to be relaxed to the original value before it can be used for the next activation. Instead, we can use the additional adder that was introduced for generating a new branch to dynamically add the noise to the weight before it is passed to the multiplier. While this does add logic to the critical path, most edge devices are already run at a much lower clock speeds [117, 118], so this will not have a significant impact on the target frequency.

*Parallel Branch Execution (AB = 00):* While the first two scenarios handled by the processing element require the extra resources allocated to properly compute the impact of noise, the execution of all in-flight branches requires only a single MAC. Adding an additional register to support having two activations arrive every cycle allows for utilization of the arithmetic resources that were allocated to handle the noise. However, feeding both of these registers at the same time would require doubling the bandwidth of the activation buffer, adding significant energy overheads for every cycle. Instead, we propose quantizing these branch activations to half the precision of the original clean activations (discussed in Section 4.4). With this representation, each memory access for all branches will inherently return two values, which can be dequantized back to the

original representation and fed directly into the extra activation register.

Each of these scenarios provides the opportunity for high resource utilization. Adding noise to an existing branch allows for fully parallel execution of two branches while only relying on a single activation. However, both parallel branch execution and adding noise to an existing branch may reduce the utilization. Adding noise to an existing branch has the worst utilization, only allowing for a single result to be calculated even though the PE has the ability to compute two. The parallel branch execution can fully feed each processing element so long as there are an even number of branches. With an odd number of branches, one cycle will only use half of the available resources. However, this slight under utilization has the potential for significant energy savings.

## 4.4 Difference Based Quantization

While the branching execution proposed in Section 4.2 provides a meaningful improvement over serial forward passes, ZO finetuning still introduces overheads for each additional branch generated. These overheads come from the additional compute and memory necessary for the additional perturbed intermediate values. The memory overhead represents a much more significant energy impact than the compute, so we try and reduce the impact of the necessary memory growth by using quantization.

Quantization, or using reduced precision values, is often difficult to integrate with training or finetuning, as the process introduces additional non-differentiable functions [119]. There are various techniques that have been developed to overcome or bypass this function [120, 113], but none recover the same accuracy as full precision quantization. Additionally, these techniques require re-training, and perform poorly when exposed to out of distribution data [121]. This causes us to avoid trying to quantize the original forward pass, maintaining full precision for the most important inference. However, we

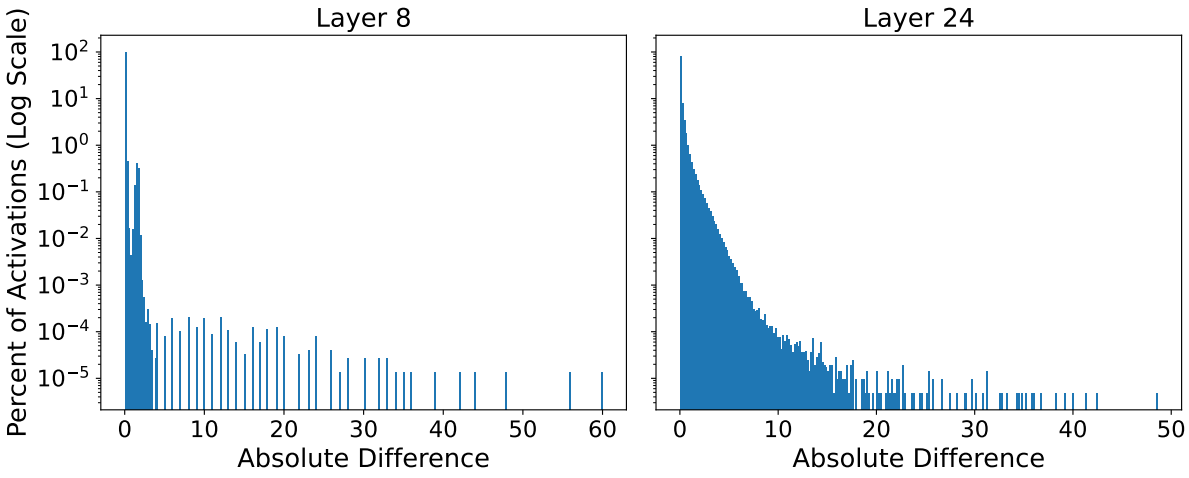



Figure 4.3: Histograms of the absolute difference between branch activations and their corresponding clean activation at two points in the network execution, one near the beginning and one at the end. The y-axis is the percentage of all activations, and is shown in a log scale.

note that the additional, perturbed runs do not necessarily need to be maintained in full precision. Many of the activations belonging to the perturbed runs will have only a small difference from the original value. Leveraging this observation, between linear layers we can store all of the perturbed runs' activations as the difference between the actual value and the non-perturbed value. Since the data is organized such that any streaming or dataflow operation will access the clean value first, it can be temporarily stored and used to recreate the original value for all branches.

By analyzing these differences, as shown in Figure 4.3, we make two observations. First, well over 90% of the values are near zero, although almost none are exactly zero. This creates an opportunity for value reuse [122, 123], where computation can be skipped by reusing the calculated result from the non-perturbed run. Second, a very small number of differences are very large, well beyond the scale that would normally be indicated by the non-perturbed value. This observation means that an overly aggressive quantization scheme cannot capture some of these outliers. For instance, these histograms use 128 bins to show the absolute value of each difference, so each bin roughly equates to how many

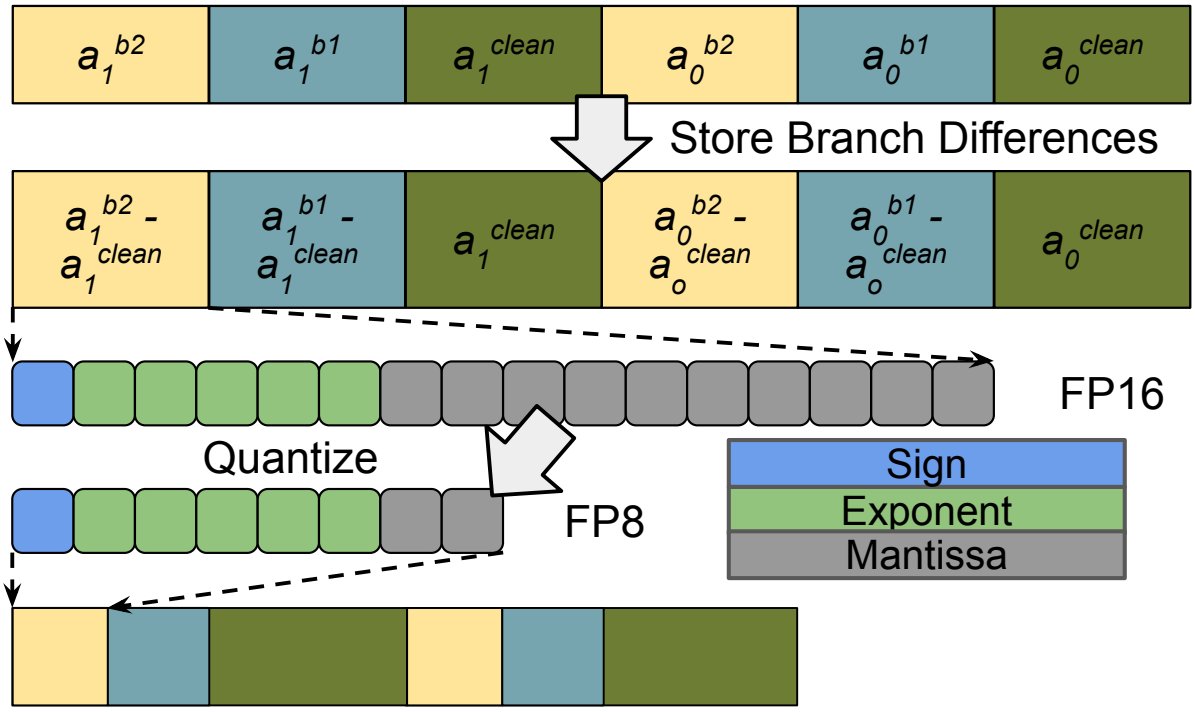



Figure 4.4: Data storage of interleaved execution. The branched activations are stored next to their corresponding clean activation. Instead of storing the true value, the difference between the branch and the clean activation is calculated, then quantized to FP8. This results in a tightly packed representation that significantly reduces the memory overhead of each additional forward pass.

values would appear at each level in an 8 bit integer or fixed point quantization scheme. This would cause almost all the values to be set to 0, destroying too much information. This motivates us to use an 8 bit floating point, which maintains good representation around 0, but can still capture the larger outliers.

Specifically we allocate 5 bits for the exponent and 2 bits for the mantissa (FP 8) as shown in Figure 4.4. This offers the large dynamic range necessary to capture outliers, and can be trivially mapped from FP 16 as they share the same exponent representation. However, using this representation makes it difficult to leverage zero-skipping, as the FP8 scheme has better resolution near zero, so many near-zero values will become zero. To combat this we introduce a scale value, common to quantization techniques [124], which

divides all of the difference values by a constant. This pushes the near zero values to be rounded down to zero, increasing our sparsity with larger scale values. Additionally, all of the large differences are pushed closer to zero where FP8 has better representation.

This allows us to halve the additional storage costs incurred by the perturbed runs. However, because our original, non-perturbed value is maintained and operated on with FP16 values, all of our arithmetic units must support FP16. Therefore, before operating on our quantized differences we convert them back into FP16. However, the original activation value very rarely needs to be fully recalculated by summing with the clean activation. For many activation functions such as SiLU and Layernorm this is the case, but for many operations we can actually decompose the computation. This relies on the same decomposition trick that we discuss in Section 4.2.1, except now the activation is decomposed:

$$O = (a^{clean} + a^{diff}) \times W = a^{clean} \times W + a^{diff} \times W$$

This means that each linear layer can be fully executed without requiring each branch activation to be summed with its corresponding clean activation.

Most executions on the systolic array can be implemented in this decomposed manner, with the only exception being the attention multiplication between the queries and keys. Since both of the incoming matrices will have branches, it would require too many extra computations to perform the matrix multiplication on just the differences. Therefore, before this query-key multiplication all of the activations are combined with the clean branch and maintained at higher precision. This causes the array utilization to be significantly worse for our proposed PE in Section 4.3 as none of the three scenarios are applicable. However, when considering edge scenarios the sequence length is always relatively small [125], causing this overhead to be fairly small relative to the entire transformer computation.

## 4.5 Architectural Support For Update Reduction

While branched and interleaved ZO training can provide significant benefits by lowering the cost of each subsequent forward pass, the memory demand is still dominated by the weights. Neural networks, including those deployed at the edge, have on the order of hundreds of millions to billions of parameters. Even with an interleaved, weight-stationary dataflow each weight must be accessed from DRAM a minimum of three times for training, first for the forward passes and then again when the projected gradient has been calculated and the weight must be read, and finally a third time when its been updated and must be written to memory. This read, update, then write introduces significant overhead, in some cases consuming more energy than a single inference.

To combat this overhead we propose a hardware-assisted weight selection process that only perturbs a limited set of parameters every linear layer. Every parameter that is not perturbed can then be skipped during the weight access and update phase, representing a significant potential savings. To do this weight selection we leverage the same insight as one commonly used in quantization and sparsity efforts: large values are important [126, 127]. Specifically, we leverage the insight that the parameters that interact with the largest activations are the ones most salient to the current computation [128]. This has previously been leveraged for channel-wise quantization, but in fine-tuning scenarios, quantization based on off-line profiling makes assumptions about the data the model will interact with. These assumptions can damage the model's flexibility and generalizability. Instead of off-line profiling, we propose a hardware block that dynamically determines the top-K largest activations entering each linear layer, then applies noise only to parameters that interact with these activations.

We also note that while the primary purpose of this technique is to reduce the overhead introduced by weight updates, it also significantly reduces the number of random

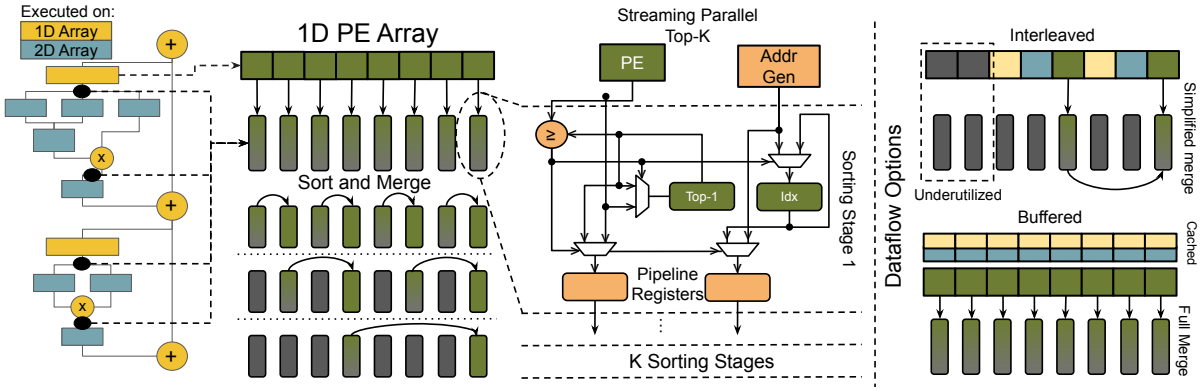



Figure 4.5: The hardware and system design for our streaming Top-K activation selection. This module only needs to be activated four times per decoding layer, with the top activations being shared across the parallel linear layers. Then, the operation of our streaming selection is fused with the 1D PE array, and perform parallel sort operations. After all of the values have been passed through the array the parallel top-K arrays must be merged to calculate the final, global top-K. The far right side shows the two potential dataflow options that can be used when considering branched activations that do not need to be sorted.

values that must be generated. This reduction in random number generations provides some energy improvements, but also serves to reduce the dimensionality of the zero order optimization. This dimensionality reduction can in some cases improve the final training performance depending on the number of branches generated during execution. We discuss this in Section 4.7.

### 4.5.1 Streaming Top-K Selection

To figure out the largest activations dynamically we use a streaming top-K selector leveraging two insights about the modern transformer. The first is that many of the linear layers share the same set of inputs. This means that the same top-K selection can be shared across all parallel linear layers. One example of this is the Query, Key and Value projections, which can share the same top-K indices. This means that each decoder layer within decode-only transformers requires only four instances of top-K selection, as shown in Figure 4.5. The second observation is that before each linear layer, or group

of parallel linear layers, is an element-wise operation. All of these operations, either a layernorm or an element-wise multiplication, will be computed on a vector processing unit, or a one dimensional compute array that is separate from the systolic array that will be used for all matrix operations.

We can directly fuse our top-K selection hardware with the 1D array, directly feeding the outputs of the array to a stream-sorting hardware unit. Assuming a fully utilized array, every cycle a new set of values will arrive at the sorting unit in a streaming manner. However, it would be cumbersome to create stream-sorting hardware that sorts every result from each PE every cycle, requiring a large number of serial comparators to determine the order of the arriving values. Additionally, these values would then need to be compared with all values currently inside the array to determine which values must be maintained. Massive single-cycle sorting like this is infeasible in hardware, so instead we turn to parallel operation. Each PE feeds its own sorting unit as shown in the middle of Figure 4.5, which only accepts a single value, and can be fully pipelined to avoid many sequential comparators. Then, after all the values have been streamed into parallel sorters and the sort pipeline settles, pairs of sorting units can be merged-in-place by streaming the contents of one sorter into another. This can then occur in a tree-based manner, requiring $\log_2(N)$ merge operations, where $N$ is the length of the 1D array.

However, this operation assumes that the top-K selection occurs for all values passing through the 1D array. Instead, the top-K value is only ever extracted from a single branch, whichever one will be perturbed at the next linear layer. This leaves two possibilities on how to manage the dataflow through the 1D array: leaving the data interleaved or buffering and unrolling the interleaved storage, both shown on the right side of Figure 4.5. Since the data is stored in an interleaved manner, all streaming 1D operations will result in the array being filled partially with data that needs to be analyzed, and some data that can be ignored. Depending on the number of branches that have been generated

fewer sorting pipelines need to be active, saving energy and significantly reducing the number of merge operations that need to occur. However, to ensure this savings the branch which is being operated on must be scheduled to the same index in the PE array every cycle. If the number of active branches do not evenly fit into the processing element ($N \mod r \neq 0$, where $r$ is the number of forward passes) then the array must be left underutilized to meet this requirement.

Alternatively, the incoming data can be buffered, then re-organized so that each cycle only a single branch is being executed on the array. This allows for full utilization of the PEs and sorting units, but requires an additional buffer that is capable of holding $N \times r_{worstcase}$. It also requires all of the sorting units to be merged, consuming the maximum possible merge cycles. This creates a tradeoff between under-utilization cycles and the merge overhead cycles, which we explore in Section 4.7.

## 4.6 Experimental Setup

In order to fully and fairly evaluate our proposed innovations we use two different pipelines, one for neural network performance evaluation and one for architectural evaluation. Across both of these frameworks we use the same model, Qwen2.5 0.5B parameter model [94] as a case study, representing a modern, small transformer model that could be deployed to resource-constrained devices. For application performance we build on the framework developed in MeZO [106], built using PyTorch and Huggingface [129], implementing our custom interleaved and branching implementation for all Qwen2.5 models. For our noise generation we use a Rademacher distribution [130] as its been demonstrated that it provides a lower variance for gradient estimation [131]. We use this framework to evaluate the accuracy impact of our proposed quantization scheme and top-K noise selection, as well as extracting the sparsity levels introduced by quantization.

For our architectural evaluation, we build on top of Timeloop [132], which analyzes the different tiling strategies available for a given architectural configuration. Using Timeloop we implement our proposed interleaved dataflow, enforcing all branched activations to immediately follow their corresponding clean activation. We analyze our proposed architecture on a 16x16 PE systolic array, a small array which can be realistically implemented on edge devices. However, we realize our proposed PE has the area overhead of two PEs, so we create a baseline architecture of 16x32 PEs to compare against. All arithmetic operations within the processing elements are modeled to operate using FP16 to match the functionality of our application evaluation system discussed above. Both our proposed array and the baseline is equipped with two 64KB SRAM buffers, one for the input activations and the other for output features. We use Accelergy [133] to evaluate the energy efficiency of our memory systems and register operations on a 40nm tech node. To model the energy consumption of the FP16 operations we use published data for 40nm arithmetic operations [134]. To analyze the latency of our proposed approach we use the cycle count estimates from Timeloop and assume a 200MHz clock with 10GB/s DRAM bandwidth. For our random number generator we used synthesized values for an 8-bit generator [11], and assume one random number generator per column in the systolic array.

To evaluate the streaming top-K module we built and verified the operation of the stream-sorting hardware in RTL, using the RTL simulation to generate a cycle accurate simulation that we integrate with our larger architectural simulator. We then synthesized the module using the Free45 PDK [135] and OpenRoad [136] to extract power, maximum frequency, and area estimates.

Table 4.1: Synthesis results for top-K streaming hardware. Includes all sorting lanes for a 32 wide PE array.

| Top-K | Power ($mW$) | Area ($mm^2$) | Max Frequency (MHz) |
|---|---|---|---|
| 5 | 15 | .074 | |
| 10 | 28.1 | .154 | 623 |
| 20 | 56.2 | .312 | |

## 4.7 Evaluation

The evaluation is split into three sections. First, we evaluate the streaming top-K selection process. The actual overhead relative to the entire network evaluation is fairly small, so the evaluation is performed relative to itself. Then we analyze the tradeoff between the number of generated branches and top-K selection on the energy and application impact of each ZO training step. Finally, we analyze the interaction between our proposed follow-along quantization, sparsity, and application performance.

### 4.7.1 Streaming Top-K Evaluation

To see the hardware overhead of our proposed top-K selection mechanism we synthesize our implemented streaming sort arrays, and report the figures of merit in Table 4.1. These synthesis results include 32 different instantiations of our proposed hardware sorters to support a 1D array with 32 PEs. The K indicates how many of the largest values each sorting buffer must maintain. From an area perspective these hardware units take up very little space, with even a top-20 sorter only consuming 0.312 $mm^2$. The power consumption has a larger impact, consuming enough energy to exceed the power budget of of the most energy constrained devices [117]. However, its worth noting that this unit is only active for a small portion of the entire training forward pass (at most 0.5% of the forward step when performing only 2 forward passes). Additionally, the utilization will always be much lower than 100%, even when activated, as only one branch per forward

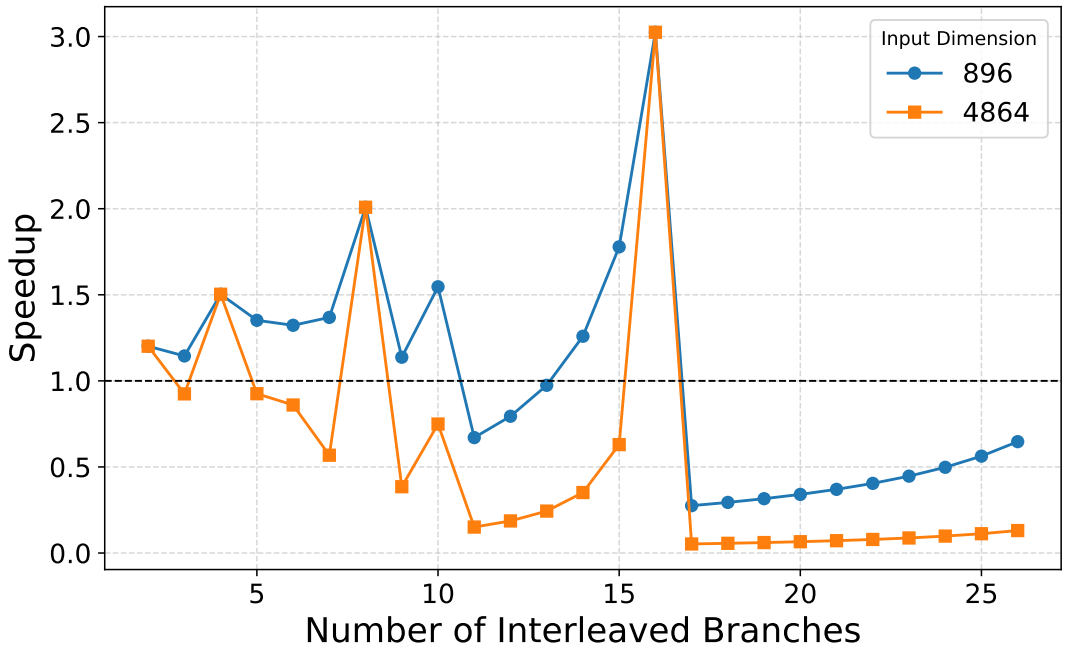



Figure 4.6: The speedup achieved by using the interleaved dataflow to reduce the number of top-K merge operations that need to occur, compared to using a buffer. The dotted line is the normalized number of cycles required when using fully buffered operations that achieves max utilization. This assumes $K = 10$.

pass will require the sorting. Therefore, regardless of which dataflow approach is used, the actual power consumption will be anywhere from one half to one tenth of what is reported here.

Next, we want to analyze the cycle overhead introduced by the top-K streaming, where we again assume a 1D array with 32 PEs. The maximum frequency will not have a large impact as most resource constrained devices operate well under 500MHz [137, 118]. Therefore, we analyze the cycle overhead in Figure 4.6 by comparing the performance of the default, interleaved execution of the 1D PE array with the potential buffered PE array, examining the trade-off between the sorting merge overhead vs the under-utilization introduced by the interleaved execution for two different values of $K$. Because buffered operation will always have the same cycle overhead (the time to fill the buffer, plus the time to fully merge all sorted arrays), it is fixed at one in this

graph. Then, the performance of the interleaved execution is shown normalized to the buffered execution time. We show how the performance changes with the number of interleaved branches on the x-axis, and each line represents the dimension of values that must be passed through the array per branch per token. In Qwen2.5, while there are four locations within the decoder layer that must be analyzed, all but one of these locations has a dimension of 896, with the last location (right before the down projection in the MLP) having a dimension of 4864.

Each peak in performance improvement occurs when the 1D PE is able to be fully utilized, when the number of branches can evenly divide the length of the PE array. This is because it allows for full utilization, while requiring fewer merge operations. However, when the number of active branches can't evenly divide the array, it results in under utilization. Sometimes this under utilization is small enough that the merge operations still take more time, but the impact is magnified when dimension being operated on is larger. However, over the course of an entire network forward pass, the interleaved approach introduces only 85.2% of the cycles introduced by a fully buffered approach.

While this can have a large impact on the local performance of the one dimensional array, it has a very small impact on the overall performance, consuming 0.1% of the execution time of the entire forward step when using fully branched execution.

### 4.7.2 Energy and Accuracy Analysis

Before being able to fully discuss the trade-offs between energy and accuracy we attempt to show how our proposed approaches impact finetuning to specific objectives. In Figure 4.7 we show this accuracy on a variety of benchmarks [95, 138]. We use a low-training sample dataset, as has been adopted when classically evaluating ZO training on transformers [106, 112, 139], where only 1000 training samples are used. For all ZO

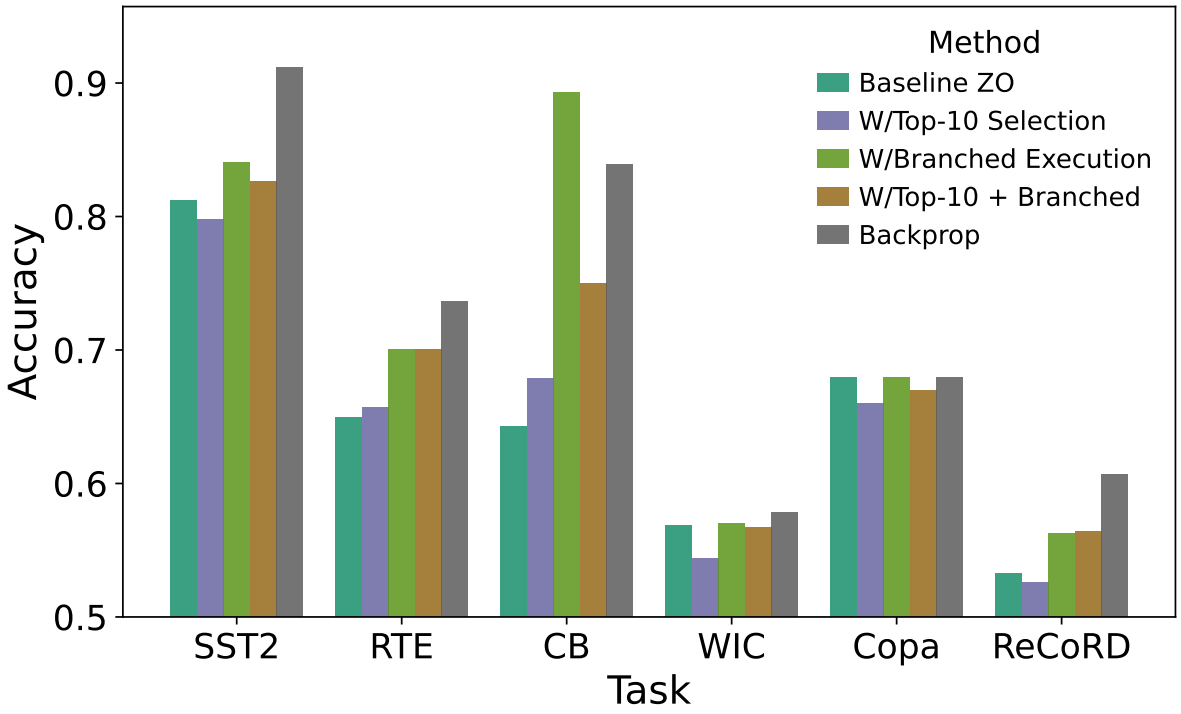



Figure 4.7: Full finetuning results across several benchmarks comparing our proposed approaches with baseline ZO training and full backpropagation with AdamW optimizer.

training techniques we use 20 thousand training steps with a learning rate of $10^{-7}$. For our branching evaluation we evaluate full branching, where each decoder layer is noised individually, generating a new branch per layer each forward pass. For our backpropagation baseline we use the AdamW optimizer [140], 1000 steps, and a learning rate of $10^{-6}$. For all training runs we use a batch size of one, emulating an edge device training as data arrives. This training scenario is unrealistic, as using 20 thousand steps is infeasible at the edge, requiring data to be captured, then constantly replayed. However, we show these results initially as it is an established approach to evaluating ZO training.

First, its noteworthy that almost all of the ZO approaches generally fall short of full backpropagation. However, as we discuss later, this backpropagation with an optimizer will always add excessive memory and energy overheads. One potential approach to avoiding this overhead is to use base stochastic gradient descent without an optimizer. However, training without an optimizer on transformer models results in incredibly poor performance. SST2 and RTE train to only 52% and 47% respectively, achieving almost no learning. However, when our approaches are compared with the baseline ZO training, a couple of notable trends appear. First, using branched execution always achieves better performance than the baseline ZO training. This is because the gradient estimation is much better, allowing for more targeted improvement. However, our top-10 parameter selection for noise injection performs inconsistently, sometimes improving on base ZO, while other times performing slightly worse. This inconsistent performance arises in part from the learning rate, which is kept fixed for all these experiments. Since fewer parameters are being updated, the overall impact to the network is smaller, preventing from improving as quickly as a full parameter update. However, this has a complex relationship with both the learning rate and the branching execution that we explore later.

To begin to see the accuracy/energy tradeoffs we first examine the energy consump-

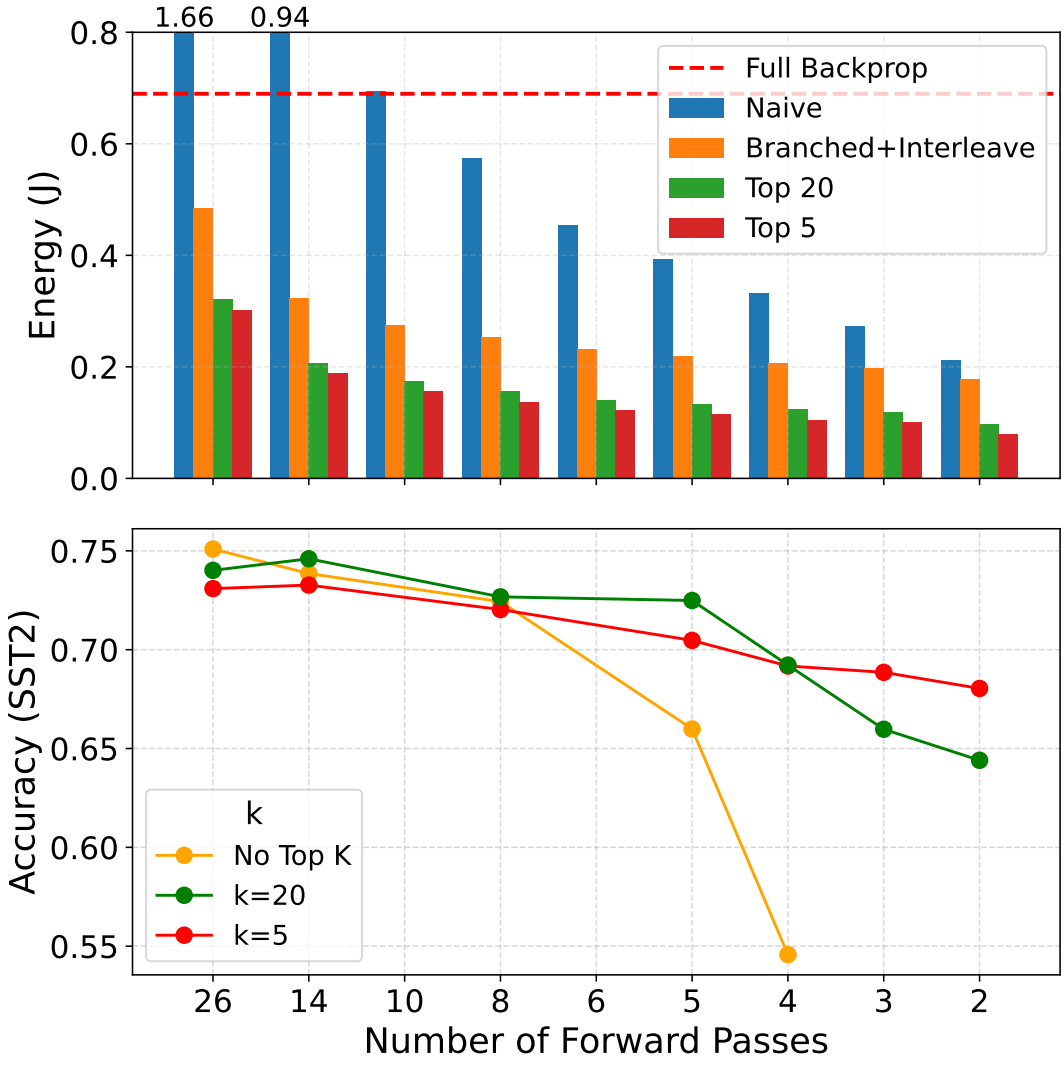



Figure 4.8: The top graph shows the full energy consumption of a single training step (forward + update) based on the number of forward passes. The dotted red line indicates the energy consumption of running full backpropagation using AdamW. The bottom graph shows the low-sample size SST2 training accuracy of different ZO training approaches based on the number of forward passes that are used. Each line represents a different top-K selection scheme. For all of our approaches, the number of forward passes refers to the number of branches when the network finishes executing.

Table 4.2: Latency, in seconds, for a single training step.

| Number of forward passes: | 26 | 14 | 8 | 4 | 2 |
|---|---|---|---|---|---|
| Naive | 3.33 | 1.79 | 1.02 | 0.51 | 0.26 |
| Branch + interleave | 0.96 | 0.61 | 0.44 | 0.32 | 0.24 |
| Top-20 | 0.99 | 0.65 | 0.47 | 0.36 | 0.24 |
| Top-5 | 0.97 | 0.63 | 0.45 | 0.35 | 0.23 |

tion, one of the largest limiting factors for resource constrained devices as they are often battery powered or have unreliable power sources. The top graph in Figure 4.8 shows the energy consumption per training step when each technique uses a different number of forward passes. The red line indicates the energy requirements for full backpropagation with an Adam optimizer, with the x-axis being the different number of forward passes generated. For this analysis we assume a small sequence length with only 16 tokens to emulate an edge device, but note that these trends hold for larger sequence lengths. The leftmost bar of each grouping represents a naive implementation of ZO training on a dedicated neural network accelerator where the forward pass is fully re-ran for every single additional forward pass. The second-from-the-left bar is a branching and interleaved execution without any of our architectural improvements. The other two bars use our proposed architecture with two different values of K for the top-K selection. For fair evaluation, our architecture uses a systolic array that is half the size of the naive and branched architectures to account for our larger PE.

The branched and interleaved execution provides the largest improvement over the naive approach to ZO training, and then our architectural improvements provide even further energy savings. Of these energy savings, a large potion comes from the update reduction, which allows for significantly fewer weights to be accessed and written, significantly reducing the number of total DRAM accesses. A smaller portion comes from our proposed processing elements and quantization, which consumes 55%-66% of the energy

of an array that is twice the size but uses standard processing elements. While the larger array can achieve better tiling strategies (at the cost of higher bandwidth SRAMs), our quantization and proposed PEs can save significantly more energy than the cost of a worse tiling strategy. While this energy savings comes at a slight cost of latency (shown in Table 4.2), this approach is still significantly faster than a naive approach to ZO training.

Compared to full backpropagation, our proposed architecture with top-20 parameter selection can achieve between 2.15× (when using full branching) and 7.07× (when only using 2 forward passes) better energy efficiency, depending on the number of forward passes being used. When compared to a naive implementation of ZO training it can achieve 2.8× better energy efficiency when using only 2 forward passes. The scheme with a maximum number of forward passes, where each decoder layer generates its own branch, can be executed with roughly the same amount of energy as running 4 forward passes in a naive manner. This shows how our architectural contributions make every additional forward pass beyond the first one significantly more energy efficient.

While its clear that the fewer forward passes, the better the energy consumption, it is unclear how this interacts with the application accuracy. To examine this, we run ZO training with a varying number of branched forward passes for different values of top-K on one of our benchmarks, SST2, shown in Figure 4.8. In this experiment we emulate a real edge-training scenario where we only allow the model to see each training sample once, so with 1000 training samples we run 1000 training steps with a batch size of 1. To fully understand the typical performance difference between each design point we run each design point 10 times and report the average of all runs. Each run uses a learning rate of $10^{-6}$ to show how our approaches can enable faster learning.

The first trend is the obvious decrease in accuracy as the number of forward passes decreases. As expected the gradient estimation gets progressively worse, so the amount

of learning that can occur with such few training steps decreases significantly. When considered in conjunction with the top graph, the takeaway is that the higher the energy budget, the better the training. However, this curve can be offset to some extent by using a smaller K. The decrease in dimensionality that comes with a decreased K does not improve accuracy when a large number of forward passes is used because the gradient estimation is already good. However, when the gradient estimation isn't as good, having fewer updates can significantly improve the accuracy. This provides significant energy savings, as fewer forward passes are required, and the number of parameters that need to be updated are also decreased.

The interactions between these two graphs reveals a complex design space, where the number of runs and the top-K have a significant impact on energy per training step as well as the amount of learning per step. To explore this we perform a large design space exploration for two different tasks, SST2 and RTE, shown in Figure 4.9. Not only do we sweep the number of forward passes per run and the top-K selection value, but we also explore the impact of the learning rate by running all design points with a learning rate of $10^{-6}$ and $10^{-7}$. The accuracy for each design point is evaluated using the same approach as the previous experiment.

These graphs illustrate both the effectiveness of increasing the number of forward passes, and how beneficial the top-K selection can be. First, the most accurate point on both graphs use the full branching method where each decoder layer generates a new branch. However, using fewer forward passes can still maintain reasonable accuracy while consuming significantly less energy. This is typically enabled by a very low top-K value, as we highlight two points along the Pareto optimal frontiers that both use K=5, the smallest value of K that we included in our sweep. While using 14 forward passes with K=5 we consume less energy than a 2 forward pass naive approach while improving on the accuracy of the naive approach by 6% and 13% for RTE and SST2 respectively. This

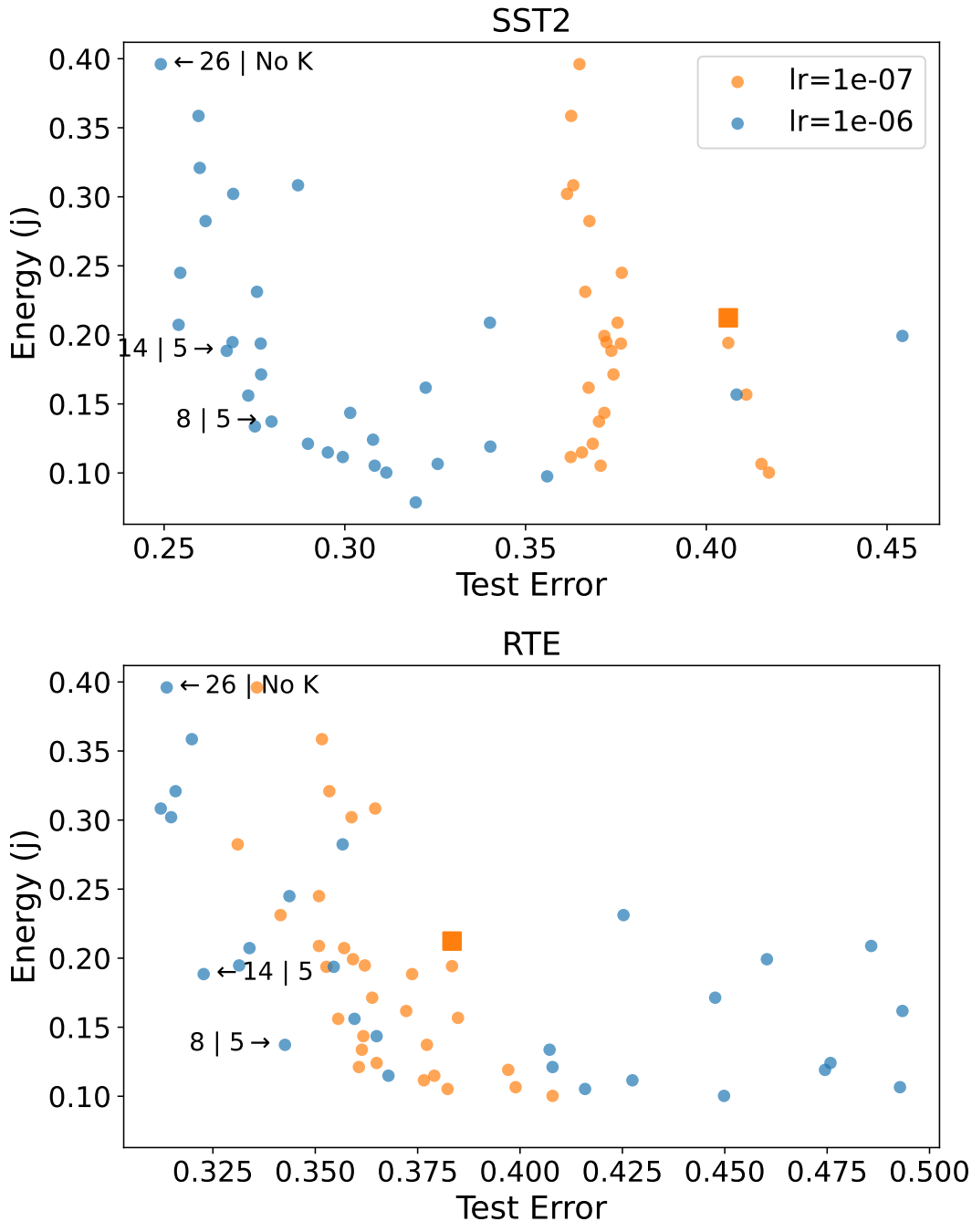



Figure 4.9: The design space exploration of energy per training step and test error after finetuning (one minus the test accuracy). The square dot represents a naive approach to ZO training where the entire model is run twice sequentially. We highlight 3 configurations, with text on the graph showing: [*number of forward passes* | $K$]

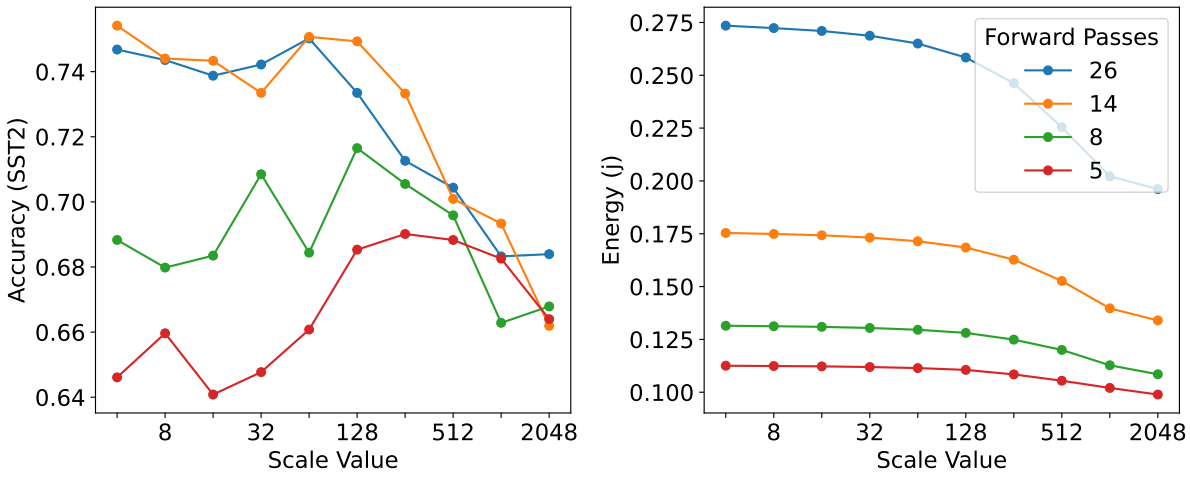



Figure 4.10: The left graph is the quantization accuracy on SST2 as the scale value in the branch quantization is increased, artificially increasing sparsity, for various number of forward passes. Similarly the right graph is how the sparsity impacts the energy consumption of the ZO inference.

accuracy improvement is in part because our approaches enable a higher learning rate, with the naive approach failing to train when using a $10^{-6}$ learning rate. Additionally, the points that use $10^{-6}$ with worse accuracy than $10^{-7}$ points stems from having a low number of forward passes and a high top-K (or no top-K). This shows how our approach can facilitate significantly faster learning.

### 4.7.3 Branch Quantization Analysis

While the above energy vs accuracy analysis does consider the impact of quantization on the reduction of energy with respect to the memory system, it does not evaluate the impact in terms of accuracy. It also does not evaluate how the sparsity introduced by quantization can save energy by using zero-skipping to avoid unnecessary arithmetic operations. To understand these impacts we run an experiment where we introduce quantization, then sweep the scale value to artificially increase sparsity. These results are shown in Figure 4.10, with the left graph showing the impact on accuracy, with a

learning rate $10^{-6}$, and the right graph showing the impact that the sparsity has on the energy consumption of the forward step for different configurations. For this experiment, we assume no top-K selection, and only run each training 5 times, as opposed to the 10 we were running previously.

The most interesting trend in terms of accuracy is how increasing the scale of the quantization, and thus the sparsity, impacts each line differently. When a large number of forward passes is used, the accuracy is fairly stable until 128. Then, for the maximum number of forward passes the accuracy starts to significantly drop after that, but when 14 forward passes are used this sharp decrease in accuracy occurs slightly later. Then, with 8 and 5 forward passes the accuracy actually increases until this point, and then decreases significantly slower. The primary reason for this is that the differences between the branch activations and the clean activations are larger when fewer branches are generated. Dividing these differences by a large scale value pushes them closer to 0, where FP8 has a much better representation.

For the energy analysis we directly extract sparsity values from running the quantized training to determine how much sparsity occurs at every layer. This layer-wise sparsity is then injected into our architectural simulator to see how much computational energy is saved. Obviously these energy savings are much larger when the number of forward passes is large, because the additional computation contributes a significant amount of overhead. However, the largest energy improvements appear between steps 128 and 1024, right when the runs with more forward passes begin to significantly drop in accuracy. However, for the runs with fewer forward passes, the quantization can reduce the energy consumption by almost 9% while actually *improving* accuracy.

## 4.8 Enhancing Temporal Computing

Another benefit of ZO optimization is its well understood ability to optimize for objectives that are non-differentiable [106]. This tracks with the behavior of ZO optimization, where the inference is performed many times, and the update is based on whether or not the objective is improved. No actual derivatives are calculated, so a metric like accuracy, which is derived through a non-differentiable function, can still be optimized. This can even be applied to non-differentiable functions within the neural network itself.

This offers a solution to the fact that our temporal approximations of arithmetic are non-differentiable. The underlying nLSE and nLDE functions have clear derivatives, with the derivative of nLSE being equal to softmin. However, the MIN, MAX, and INHIBIT operations used to approximate these operations are all non-differentiable. This is not a major issue when trying to find the optimal weights to pair with the approximations. A training loop can be set up that performs the forward pass with the approximations, then calculates the gradients as if full precision importance space operations were used. This approximation-in-the-loop approach shares many similarities to straight through gradient estimators that are commonly used when training quantized neural networks [120]. However, this approach is incapable of trying to optimize the constants used for the approximation terms (C & D in Equation 2.6 and E & F in Equation 2.7).

These approximation constants are well suited for ZO optimization, as they have a large impact on the accuracy of the neural network while still having a relatively small number of parameters that need to be changed. To show the potential benefits of this combination we present an experiment performed on the Fasion MNIST dataset [141] where we examine both the ZO optimization of the constants and the approximation-in-the-loop training. The results of this experiment is shown in Table 4.3.

These results show that ZO optimization can be used to improve the accuracy of the

| Approximation Terms | Baseline | Constants ZO | Approximations-in-the-loop | Both |
|---|---|---|---|---|
| 1 | 35.97% | 53.08% | 64.00% | 66.71% |
| 2 | 66.14% | 68.92% | 78.95% | 80.04% |
| 3 | 72.24% | 78.40% | 82.00% | 82.49% |
| 4 | 77.92% | 80.62% | 83.76% | 84.56% |
| 5 | 80.42% | 83.29% | 84.67% | 84.86% |

Table 4.3: Analysis of ZO optimization of the approximation constants and the approximation-in-the-loop training on a 2 layer network on Fashion MNIST. The original network was trained without regularization, and the approximation constants are initialized using the technique discussed in Chapter 2. The baseline network has an accuracy of 87.24% when executed in importance space, and each training approach uses 5 epochs. The approach that uses both uses 5 epochs for approximations-in-the-loop training, then 5 epochs of ZO training.

temporal neural networks. When the number of approximations terms are small it can provide a significant benefit, improving the 1 approximation term system by roughly 17%, taking the network from unusable to a network that is beginning to have some utility. While this improvement slows as the number of approximation terms increases, this is to be expected, as the approximation is getting better, and cannot be improved significantly. However, even at 5 approximation terms there is an almost 3% improvement in accuracy. The approximations-in-the-loop training approach out performs the ZO optimization in all scenarios, but this makes sense as it optimizes the network weights, which has a significantly higher number of parameters than just the approximation constants. While this training shows significant improvement over the baseline, it can still be improved slightly by also incorporating the ZO optimization, with each approximation term having slightly better accuracy when both techniques are used.

## 4.9 Conclusion

In this chapter we present a novel architecture and execution approach to accelerate ZO finetuning on transformer models. Unlike traditional training this approach

requires no expensive backpropagation or optimizers, making it an attractive alternative for resource constrained devices. We show how a tailored architecture can parallelize the multiple forward passes necessary when computing a ZO training step, significantly reducing the energy and delay overhead. We also demonstrate architectural optimizations that enables faster learning while simultaneously improving the energy efficiency of the system. All of these proposed innovations taken together creates a rich design space that we thoroughly explore, demonstrating a clear energy/accuracy tradeoff. Then we demonstrate how this ZO optimization can be applied to non-transformer temporal neural networks, creating an approach that is capable of improving base neural network accuracy.

# Chapter 5

# Conclusions and Outlook

The combination of significantly slowed semiconductor scaling and a massive increase in the demand for new computation has created a need for alternative low-power solutions. Temporal computation is one such alternative, relying on a form of computation with a very low activity factor that can directly integrate into some of the largest data-producing devices: sensors. This offers energy savings directly through several different mechanisms: reducing expensive domain conversions, using low-energy arithmetic, and using single-cell memory devices. These direct energy savings can also translate to indirect energy savings, as neural network operation can begin to push toward the edge, removing the constant communication and connection overheads that accompany cloud-based systems.

To realize the potential of temporal computing, this thesis proposes several innovations. The negative log-transformed arithmetic and approximations enable a much wider range of applications, creating utility far beyond that of the tropical algebra problems that have previously been mapped to temporal computations. This arithmetic, combined with recurrence as a form of short-term memory and emerging devices that serve as long-term memory, enables programmable architectures capable of very energy-efficient neural networks with minimal algorithmic changes. While these neural networks achieve suffi-

cient accuracy without any additional training, ZO optimization can be used to improve this accuracy while also providing a mechanism for energy-efficient fine-tuning of traditional digital neural networks.

### 5.0.1 Future Work

The temporal computing framework, as described in this thesis, still has many limitations. The ability to implement full transformer models, the basis of modern AI, is limited without additional hardware, which introduces significant overhead such that the energy efficiency is lost. To overcome these overheads, innovations across the computing stack are necessary, requiring neural network models to be co-designed alongside the temporal hardware. However, this framework was inspired by the spike-based computation used in biological systems, so model development that is aware of the limitations of temporal representations may help create even more biologically plausible neural networks.

Additionally, ZO optimization can benefit from temporal computation beyond the connection discussed in this dissertation. ZO optimization moves the bottleneck of training from memory-dominant backpropagation to computation, which can be made significantly more efficient with temporal arithmetic. The interleaved execution of the forward passes would involve repeated temporal pulses, creating waves of signals and introducing frequency information that may be leveraged as an additional dimension for information or explainability.

ZO optimization also has interesting architectural research avenues beyond temporal computing, introducing interesting interactions with standard digital computing. The difference-based quantization discussed in this thesis can also be applied when the original network is quantized, translating the minor differences between forward passes into values with very few non-zero bits. When several forward passes are interleaved together, this

creates opportunities for aggressive value reuse that requires minimal activity factors for every repeated computation.

# Bibliography


[1] Q. Cao, X. Chen, and J. Gu, *Development of tropical algebraic accelerator with energy efficient time-domain computing for combinatorial optimization and machine learning*, in *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2023.

[2] A. Madhavan, T. Sherwood, and D. Strukov, *Race logic: A hardware acceleration for dynamic programming algorithms*, *ACM SIGARCH Computer Architecture News* **42** (2014), no. 3 517–528.

[3] G. Tzimpragos, A. Madhavan, D. Vasudevan, D. Strukov, and T. Sherwood, *Boosted race trees for low energy classification*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 215–228, 2019.

[4] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, *Stacked convolutional auto-encoders for hierarchical feature extraction*, in *21st International Conference on Artificial Neural Networks (ICANN)*, pp. 52–59, Springer, 2011.

[5] S. J. Sangwine and T. A. Ell, *Colour image filters based on hypercomplex convolution*, *IEE Proceedings-Vision, Image and Signal Processing* **147** (2000), no. 2 89–93.

[6] D. Ziou, S. Tabbone, *et. al.*, *Edge detection techniques-an overview*, *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii* **8** (1998) 537–559.

[7] D. G. Costa, *Visual sensors hardware platforms: A review*, *IEEE Sensors Journal* **20** (2020), no. 8 4025–4033.

[8] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, *A survey of convolutional neural networks: analysis, applications, and prospects*, *IEEE transactions on neural networks and learning systems* (2021).

[9] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks: A tutorial and survey*, *Proceedings of the IEEE* **105** (2017), no. 12 2295–2329.

[10] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, *et. al.*, *A study of BFLOAT16 for deep learning training*, *arXiv preprint arXiv:1905.12322* (2019).

[11] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. San Miguel, *UGEMM: Unary computing architecture for GEMM applications*, in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 377–390, IEEE, 2020.

[12] D. Wu and J. San Miguel, *uSystolic: Byte-crawling unary systolic array*, in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 12–24, IEEE, 2022.

[13] A. Madhavan, T. Sherwood, and D. Strukov, *Race logic: A hardware acceleration for dynamic programming algorithms*, in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, ISCA '14, p. 517–528, IEEE Press, 2014.

[14] J. Smith, *Space-time algebra: A model for neocortical computation*, in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 289–300, IEEE, 2018.

[15] A. Madhavan, T. Sherwood, and D. Strukov, *A 4-mm2 180-nm-CMOS 15-giga-cell-updates-per-second DNA sequence alignment engine based on asynchronous race conditions*, in *IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, 2017.

[16] N. Mahapatra, A. Tareen, and S. Garimella, *Comparison and analysis of delay elements*, in *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, vol. 2, pp. II–II, 2002.

[17] J. S. Bridle, *Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition*, in *Neurocomputing*, (Berlin, Heidelberg), pp. 227–236, Springer Berlin Heidelberg, 1990.

[18] M. L. Bynum, G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff, *Pyomo–Optimization modeling in Python*, vol. 67. Springer Science & Business Media, third ed., 2021.

[19] R. H. Byrd, J. Nocedal, and R. Waltz, *KNITRO: An integrated package for nonlinear optimization*. Large-Scale Nonlinear Optimization. Springer, 2006.

[20] W. Cao, X. He, A. Chakrabarti, and X. Zhang, *NeuADC: Neural network-inspired RRAM-based synthesizable analog-to-digital conversion with reconfigurable quantization support*, in *Design, Automation and Test in Europe Conference (DATE)*, pp. 1477–1482, 2019.

[21] G. Tzimpragos, J. Volk, A. Wynn, J. E. Smith, and T. Sherwood, *Superconducting computing with alternating logic elements*, in *International Symposium on Computer Architecture (ISCA)*, pp. 651–664, IEEE, 2021.

[22] A. Madhavan, M. W. Daniels, and M. D. Stiles, *Temporal state machines: Using temporal memory to stitch time-based graph computations*, *J. Emerg. Technol. Comput. Syst.* **17** (may, 2021).

[23] H. Vakili, M. N. Sakib, S. Ganguly, M. Stan, M. W. Daniels, A. Madhavan, M. D. Stiles, and A. W. Ghosh, *Temporal memory with magnetic racetracks*, *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* **6** (2020), no. 2 107–115.

[24] A. Madhavan and M. D. Stiles, *Storing and retrieving wavefronts with resistive temporal memory*, in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[25] A. Sayal, S. Fathima, S. T. Nibhanupudi, and J. P. Kulkarni, *COMPAC: Compressed time-domain, pooling-aware convolution CNN engine with reduced data movement for energy-efficient AI computing*, *IEEE Journal of Solid-State Circuits* **56** (2021), no. 7 2205–2220.

[26] G. C. Holst and T. S. Lomheim, *CMOS/CCD sensors and camera systems*. SPIE Press Monograph, 2007.

[27] H. Mostafa and Y. I. Ismail, *Highly-linear voltage-to-time converter (VTC) circuit for time-based analog-to-digital converters (T-ADCs)*, in *IEEE 20th international conference on electronics, circuits, and systems (ICECS)*, pp. 149–152, IEEE, 2013.

[28] H. Pekau, A. Yousif, and J. W. Haslett, *A cmos integrated linear voltage-to-pulse-delay-time converter for time based analog-to-digital converters*, in *IEEE International Symposium on Circuits and Systems*, pp. 4–pp, IEEE, 2006.

[29] A. H. Hassan, H. Mostafa, T. Ismail, and S. Gabran, *An ultra-low power voltage-to-time converter (VTC) circuit for low power and low speed applications*, in *29th IEEE international system-on-chip conference (SOCC)*, pp. 178–182, IEEE, 2016.

[30] A. Elgreatly, A. Dessouki, H. Mostafa, R. Abdalla, and E.-s. El-Rabaie, *A novel highly linear voltage-to-time converter (VTC) circuit for time-based analog-to-digital converters (ADC) using body biasing*, *Electronics* **9** (2020), no. 12 2033.

[31] X. Mo, J. Wu, N. Wary, and T. C. Carusone, *Design methodologies for low-jitter CMOS clock distribution*, *IEEE Open Journal of the Solid-State Circuits Society* **1** (2021) 94–103.

[32] W. Zhao and Y. Cao, *New generation of predictive technology model for sub-45 nm early design exploration*, *IEEE Transactions on electron Devices* **53** (2006), no. 11 2816–2823.

[33] J. Howard, "Imagenette." https://github.com/fastai/imagenette/.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *ImageNet: A large-scale hierarchical image database*, in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[35] G. Bradski, *The OpenCV Library*, *Dr. Dobb's Journal of Software Tools* (2000).

[36] M. Lefebvre, L. Moreau, R. Dekimpe, and D. Bol, *7.7 a 0.2-to-3.6TOPS/W programmable convolutional imager SoC with in-sensor current-domain ternary-weighted MAC operations for feature extraction and region-of-interest detection*, in *IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, pp. 118–120, 2021.

[37] R. Enomoto, T. Iizuka, T. Koga, T. Nakura, and K. Asada, *A 16-bit 2.0-ps resolution two-step TDC in 0.18-$\mu$ m CMOS utilizing pulse-shrinking fine stage with built-in coarse gain calibration*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **27** (2018), no. 1 11–19.

[38] P. W. Fry, P. J. Noble, and R. J. Rycroft, *Fixed-pattern noise in photomatrices*, *IEEE Journal of Solid-State Circuits* **5** (1970), no. 5 250–254.

[39] O. E. Dial, *Ccd performance model*, in *Surveillance Technologies*, vol. 1479, pp. 2–11, SPIE, 1991.

[40] A. Theuwissen, *There's more to the picture than meets the eye, and in the future it will only become more so*, in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 30–35, 2021.

[41] Q. Zhang, S. Su, and M. S.-W. Chen, *A cost-efficient fully synthesizable stochastic time-to-digital converter design based on integral nonlinearity scrambling*, in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1021–1026, 2022.

[42] R. Graça, B. McReynolds, and T. Delbruck, *Shining light on the dvs pixel: A tutorial and discussion about biasing and optimization*, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4045–4053, 2023.

[43] R. Whyte, L. Streeter, M. J. Cree, and A. A. Dorrington, *Application of lidar techniques to time-of-flight range imaging*, *Applied optics* **54** (2015), no. 33 9654–9664.

[44] G. Shen, D. Zhao, T. Li, J. Li, and Y. Zeng, *Are conventional snns really efficient? a perspective from network quantization*, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 27538–27547, June, 2024.

[45] Y. Hu, Q. Zheng, G. Li, H. Tang, and G. Pan, *Toward large-scale spiking neural networks: A comprehensive survey and future directions*, *arXiv preprint arXiv:2409.02111* (2024).

[46] J. E. Pedersen, S. Abreu, M. Jobst, G. Lenz, V. Fra, F. C. Bauer, D. R. Muir, P. Zhou, B. Vogginger, K. Heckel, G. Urgese, S. Shankar, T. C. Stewart, S. Sheik, and J. K. Eshraghian, *Neuromorphic intermediate representation: A unified instruction set for interoperable brain-inspired computing*, *Nature Communications* **15** (Sept., 2024) 8122.

[47] A. Madhavan and M. D. Stiles, *Storing and retrieving wavefronts with resistive temporal memory*, in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2020.

[48] R. Gretsch, P. Song, A. Madhavan, J. Lau, and T. Sherwood, *Energy efficient convolutions with temporal arithmetic*, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 354–368, 2024.

[49] R. Brette, *Philosophy of the spike: rate-based vs. spike-based theories of the brain*, *Frontiers in systems neuroscience* **9** (2015) 151.

[50] J. Smith, *Space-time algebra: A model for neocortical computation*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 289–300, 2018.

[51] A. F. Agarap, *Deep learning using rectified linear units (ReLU)*, *arXiv preprint arXiv:1803.08375* (2018).

[52] A. Krogh and J. Hertz, *A simple weight decay can improve generalization*, *Advances in neural information processing systems* **4** (1991).

[53] K. Audhkhasi, O. Osoba, and B. Kosko, *Noise-enhanced convolutional neural networks*, *Neural Networks* **78** (2016) 15–23.

[54] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: a simple way to prevent neural networks from overfitting*, *The journal of machine learning research* **15** (2014), no. 1 1929–1958.

[55] W. Wilson, T. Chen, and R. Selby, *A current-starved inverter-based differential amplifier design for ultra-low power applications*, in *2013 IEEE 4th Latin American symposium on circuits and systems (LASCAS)*, pp. 1–4, IEEE, 2013.

[56] M. Maymandi-Nejad and M. Sachdev, *A digitally programmable delay element: design and analysis*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **11** (2003), no. 5 871–878.

[57] A. Mal, A. K. Dwivedi, and A. Islam, *A comparative analysis of various programmable delay elements using predictive technology model*, in *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*, pp. 1–5, 2016.

[58] Z. Chen and J. Gu, *Analysis and design of energy efficient time domain signal processing*, in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 100–105, 2016.

[59] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, *A white paper on neural network quantization*, *arXiv preprint arXiv:2106.08295* (2021).

[60] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: Principles and empirical evaluation*, *arXiv preprint arXiv:2004.09602* (2020).

[61] J. Zhao, S. Dai, R. Venkatesan, B. Zimmer, M. Ali, M.-Y. Liu, B. Khailany, W. J. Dally, and A. Anandkumar, *Lns-madam: Low-precision training in logarithmic number system using multiplicative weight update*, *IEEE Transactions on Computers* **71** (2022), no. 12 3179–3190.

[62] D. Miyashita, E. H. Lee, and B. Murmann, *Convolutional neural networks using logarithmic data representation*, *arXiv preprint arXiv:1603.01025* (2016).

[63] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, *SCALE-Sim: Systolic CNN accelerator simulator*, *arXiv preprint arXiv:1811.02883* (2018).

[64] T. Xanthopoulos, *Clocking in modern VLSI systems*. Springer Science & Business Media, 2009.

[65] E. Kadric, P. Gurniak, and A. DeHon, *Accurate parallel floating-point accumulation*, *IEEE Transactions on Computers* **65** (2016), no. 11 3224–3238.

[66] Y.-H. 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** (2016), no. 1 127–138.

[67] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et. al.*, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[68] A. Agrawal, S. K. Lee, J. Silberman, M. M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. M. Fleischer, M. Guillorn, M. Cohen, S. M. Mueller, J. Oh, M. Lutz, J. Jung, S. O. Koswatta, C. Zhou, V. Zalani, J. Bonanno, R. Casatuta, C.-Y. Chen, J. Choi, H. Haynie, A. Herbert, R. Jain, M. Kar, K.-H. Kim, Y. Li, Z. Ren, S. Rider, M. Schaal, K. Schelm, M. Scheuermann, X. Sun, H. Tran, N. Wang, W. Wang, X. Zhang, V. Shah, B. W. Curran, V. Srinivasan, P.-F. Lu, S. Shukla, L. Chang, and K. Gopalakrishnan, *A 7nm 4-core AI chip with 25.6TFLOPS hybrid FP8 training, 102.4TOPS INT4 inference and workload-aware throttling*, *2021 IEEE International Solid- State Circuits Conference (ISSCC)* **64** (2021) 144–146.

[69] H. Vakili, M. N. Sakib, S. Ganguly, M. Stan, M. W. Daniels, A. Madhavan, M. D. Stiles, and A. W. Ghosh, *Temporal memory with magnetic racetracks*, *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* **6** (2020), no. 2 107–115.

[70] J. Riquelme and I. Vourkas, *A star network of bipolar memristive devices enables sensing and temporal computing*, *Sensors* **24** (2024), no. 2 512.

[71] M. N. Sakib, R. Sreekumar, V. Verma, T. Tracy, and M. R. Stan, *Atcpim: Analog to time coded processing in memory for iot at the edge*, in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pp. 704–709, IEEE, 2021.

[72] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, *The missing memristor found*, *nature* **453** (2008), no. 7191 80–83.

[73] C. Li, M. Hu, Y. Li, H. Jiang, N. Ge, E. Montgomery, J. Zhang, W. Song, N. Dávila, C. E. Graves, *et. al.*, *Analogue signal and image processing with large memristor crossbars*, *Nature electronics* **1** (2018), no. 1 52–59.

[74] A. Madhavan, M. W. Daniels, and M. D. Stiles, *Temporal state machines: Using temporal memory to stitch time-based graph computations*, *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **17** (2021), no. 3 1–27.

[75] R. Yin, A. Moitra, A. Bhattacharjee, Y. Kim, and P. Panda, *SATA: Sparsity-aware training accelerator for spiking neural networks*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **42** (2023), no. 6 1926–1938.

[76] S. Lym and M. Erez, *FlexSA: Flexible systolic array architecture for efficient pruned dnn model training*, *arXiv preprint arXiv:2004.13027* (2020).

[77] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[78] X. Mo, J. Wu, N. Wary, and T. C. Carusone, *Design methodologies for low-jitter CMOS clock distribution*, *IEEE Open Journal of the Solid-State Circuits Society* **1** (2021) 94–103.

[79] A. Dehghani, M. Saneei, and A. Mahani, *A high-resolution time-to-digital converter using a three-level resolution*, *International Journal of Electronics* **103** (2016), no. 8 1248–1261.

[80] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, *A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim*, in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 58–68, IEEE, 2020.

[81] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, *CACTI 7: New tools for interconnect exploration in innovative off-chip memories*, *ACM Transactions on Architecture and Code Optimization (TACO)* **14** (2017), no. 2 1–25.

[82] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, *Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **31** (2012), no. 7 994–1007.

[83] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. Di Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, *MLPerf Tiny Benchmark*, 2021.

[84] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.

[85] S. Sarangi and B. Baas, *DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era*, in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2021.

[86] A. G. Howard, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, *arXiv preprint arXiv:1704.04861* (2017).

[87] A. Madhavan, T. Sherwood, and D. Strukov, *A 4-mm 2 180-nm-CMOS 15-giga-cell-updates-per-second DNA sequence alignment engine based on asynchronous race conditions*, in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, IEEE, 2017.

[88] Z. Chen and J. Gu, *High-throughput dynamic time warping accelerator for time-series classification with pipelined mixed-signal time-domain computing*, *IEEE Journal of Solid-State Circuits* **56** (2020), no. 2 624–635.

[89] Z. Chen and J. Gu, *19.7 a scalable pipelined time-domain DTW engine for time-series classification using multibit time flip-flops with 140giga-cell-updates/s throughput*, in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 324–326, IEEE, 2019.

[90] A. Sayal, S. Fathima, S. T. Nibhanupudi, and J. P. Kulkarni, *Compac: Compressed time-domain, pooling-aware convolution CNN engine with reduced data movement for energy-efficient AI computing*, *IEEE Journal of Solid-State Circuits* **56** (2020), no. 7 2205–2220.

[91] N. Jourdan, S. Sen, E. J. Husom, E. Garcia-Ceja, T. Biegel, and J. Metternich, *On the reliability of machine learning applications in manufacturing environments*, *arXiv preprint arXiv:2112.06986* (2021).

[92] G. J. Bowden, H. R. Maier, and G. C. Dandy, *Real-time deployment of artificial neural network forecasting models: Understanding the range of applicability*, *Water Resources Research* **48** (2012), no. 10.

[93] N. Jourdan, T. Bayer, T. Biegel, and J. Metternich, *Handling concept drift in deep learning applications for process monitoring*, *Procedia CIRP* **120** (2023) 33–38.

[94] A. Yang, B. Yu, C. Li, D. Liu, F. Huang, H. Huang, J. Jiang, J. Tu, J. Zhang, J. Zhou, *et. al.*, *Qwen2. 5-1m technical report*, *arXiv preprint arXiv:2501.15383* (2025).

[95] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, *Recursive deep models for semantic compositionality over a sentiment treebank*, in *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013.

[96] F. Hinder, V. Vaquet, and B. Hammer, *One or two things we know about concept drift—a survey on monitoring in evolving environments. part a: detecting concept drift*, *Frontiers in Artificial Intelligence* **7** (2024) 1330257.

[97] S. Greco, B. Vacchetti, D. Apiletti, and T. Cerquitelli, *Unsupervised concept drift detection from deep learning representations in real-time*, *IEEE Transactions on Knowledge and Data Engineering* (2025).

[98] R. Rojas, *The backpropagation algorithm*, in *Neural networks: a systematic introduction*, pp. 149–182. Springer, 1996.

[99] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, *Carbon emissions and large neural network training*, *arXiv preprint arXiv:2104.10350* (2021).

[100] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, *A domain-specific supercomputer for training deep neural networks*, *Communications of the ACM* **63** (2020), no. 7 67–78.

[101] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[102] G. Hinton, *The forward-forward algorithm: Some preliminary investigations*, *arXiv preprint arXiv:2212.13345* **2** (2022), no. 3 5.

[103] J. C. Spall, *Multivariate stochastic approximation using a simultaneous perturbation gradient approximation*, *IEEE transactions on automatic control* **37** (2002), no. 3 332–341.

[104] M. Gutmann and A. Hyvärinen, *Noise-contrastive estimation: A new estimation principle for unnormalized statistical models*, in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 297–304, JMLR Workshop and Conference Proceedings, 2010.

[105] S. Liu, P.-Y. Chen, B. Kailkhura, G. Zhang, A. O. Hero III, and P. K. Varshney, *A primer on zeroth-order optimization in signal processing and machine learning: Principals, recent advances, and applications*, *IEEE Signal Processing Magazine* **37** (2020), no. 5 43–54.

[106] S. Malladi, T. Gao, E. Nichani, A. Damian, J. D. Lee, D. Chen, and S. Arora, *Fine-tuning language models with just forward passes*, *Advances in Neural Information Processing Systems* **36** (2023) 53038–53075.

[107] Y. Yang, K. Zhen, E. Banijamali, A. Mouchtaris, and Z. Zhang, *Adazeta: Adaptive zeroth-order tensor-train adaption for memory-efficient large language models fine-tuning*, in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 977–995, 2024.

[108] A. Agarwal, M. J. Wainwright, P. Bartlett, and P. Ravikumar, *Information-theoretic lower bounds on the oracle complexity of convex optimization*, *Advances in Neural Information Processing Systems* **22** (2009).

[109] A. Chen, Y. Zhang, J. Jia, J. Diffenderfer, J. Liu, K. Parasyris, Y. Zhang, Z. Zhang, B. Kailkhura, and S. Liu, *Deepzero: Scaling up zeroth-order optimization for deep model training*, *arXiv preprint arXiv:2310.02025* (2023).

[110] J. C. Spall, *Multivariate stochastic approximation using a simultaneous perturbation gradient approximation*, *IEEE transactions on automatic control* **37** (2002), no. 3 332–341.

[111] Y. Zhao, H. Li, I. Young, and Z. Zhang, *Poor man's training on mcus: A memory-efficient quantized back-propagation-free approach*, *ACM Transactions on Design Automation of Electronic Systems* **30** (2025), no. 5 1–33.

[112] Y. Yang, Z. Zhang, R. V. Swaminathan, J. Liu, N. Susanj, and Z. Zhang, *Sharpzo: Hybrid sharpness-aware vision language model prompt tuning via forward-only passes*, *arXiv preprint arXiv:2506.20990* (2025).

[113] J. Zhou, Y. Yang, K. Zhen, Z. Liu, Y. Zhao, E. Banijamali, A. Mouchtaris, N. Wong, and Z. Zhang, *QuZO: Quantized zeroth-order fine-tuning for large language models*, in *Proc. Conference on Empirical Methods in Natural Language Processing*, (Suzhou, China), Nov., 2025.

[114] K. G. Jamieson, R. Nowak, and B. Recht, *Query complexity of derivative-free optimization*, *Advances in Neural Information Processing Systems* **25** (2012).

[115] A. S. Berahas, L. Cao, K. Choromanski, and K. Scheinberg, *A theoretical and empirical comparison of gradient approximations in derivative-free optimization*, *Foundations of Computational Mathematics* **22** (2022), no. 2 507–560.

[116] Z. Zhang, Y. Yang, K. Zhen, N. Susanj, A. Mouchtaris, S. Kunzmann, and Z. Zhang, *Mazo: Masked zeroth-order optimization for multi-task fine-tuning of large language models*, *arXiv preprint arXiv:2502.11513* (2025).

[117] K. Sriram, I. Karageorgos, X. Wen, J. Veselỳ, N. Lindsay, M. Wu, L. Khazan, R. P. Pothukuchi, R. Manohar, and A. Bhattacharjee, *Halo: A hardware–software co-designed processor for brain–computer interfaces*, *Ieee micro* **43** (2023), no. 3 64–72.

[118] J. Dumoulin, P. Houshmand, V. Jain, and M. Verhelst, *Enabling efficient hardware acceleration of hybrid vision transformer (vit) networks at the edge*, in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2024.

[119] S. Chen, W. Wang, and S. J. Pan, *Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization*, *Advances in Neural Information Processing Systems* **32** (2019).

[120] Z. Liu, K.-T. Cheng, D. Huang, E. P. Xing, and Z. Shen, *Nonuniform-to-uniform quantization: Towards accurate quantization via generalized straight-through estimation*, in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 4942–4952, 2022.

[121] Z. Deng, G. Chen, S. Niu, H. Luo, S. Zhang, Y. Yang, R. Chen, W. Luo, and M. Tan, *Test-time model adaptation for quantized neural networks*, in *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 7258–7267, 2025.

[122] Z. Pan, J. San Miguel, and D. Wu, *Carat: Unlocking value-level parallelism for multiplier-free gemms*, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 167–184, 2024.

[123] C. Guo, C. Wei, J. Tang, B. Duan, S. Han, H. Li, and Y. Chen, *Transitive array: An efficient gemm accelerator with result reuse*, in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pp. 990–1004, 2025.

[124] R. Krishnamoorthi, *Quantizing deep convolutional networks for efficient inference: A whitepaper*, *arXiv preprint arXiv:1806.08342* (2018).

[125] K. Wu, J. Zhang, H. Peng, M. Liu, B. Xiao, J. Fu, and L. Yuan, *Tinyvit: Fast pretraining distillation for small vision transformers*, in *European conference on computer vision*, pp. 68–85, Springer, 2022.

[126] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, *A simple and effective pruning approach for large language models*, *arXiv preprint arXiv:2306.11695* (2023).

[127] J. Frankle and M. Carbin, *The lottery ticket hypothesis: Finding sparse, trainable neural networks*, *arXiv preprint arXiv:1803.03635* (2018).

[128] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, *Awq: Activation-aware weight quantization for on-device llm compression and acceleration*, *Proceedings of machine learning and systems* **6** (2024) 87–100.

[129] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, *Huggingface's transformers: State-of-the-art natural language processing*, 2020.

[130] S. J. Montgomery-Smith, *The distribution of rademacher sums*, *Proceedings of the American Mathematical Society* **109** (1990), no. 2 517–522.

[131] S. Ma and H. Huang, *Revisiting zeroth-order optimization: Minimum-variance two-point estimators and directionally aligned perturbations*, in *The Thirteenth International Conference on Learning Representations*, 2025.

[132] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, *Timeloop: A systematic approach to dnn accelerator evaluation*, in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315, IEEE, 2019.

[133] Y. N. Wu, J. S. Emer, and V. Sze, *Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs*, in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2019.

[134] M. Horowitz, *1.1 computing's energy problem (and what we can do about it)*, in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[135] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, *et. al.*, *Freepdk: An open-source variation-aware design kit*, in *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*, pp. 173–174, IEEE, 2007.

[136] T. Ajayi and D. Blaauw, *Openroad: Toward a self-driving, open-source digital layout implementation tool chain*, in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019.

[137] D. Garrett, Y. S. Park, S. Kim, J. Sharma, W. Huang, M. Shaghaghi, V. Parthasarathy, S. Gibellini, S. Bailey, M. Moturi, *et. al.*, *A 1mw always-on computer vision deep learning neural decision processor*, in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 8–10, IEEE, 2023.

[138] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, *Glue: A multi-task benchmark and analysis platform for natural language understanding*, in *Proceedings of the 2018 EMNLP workshop BlackboxNLP: Analyzing and interpreting neural networks for NLP*, pp. 353–355, 2018.

[139] L. Gao, A. Ziashahabi, Y. Niu, S. Avestimehr, and M. Annavaram, *Enabling resource-efficient on-device fine-tuning of llms using only inference engines*, *arXiv preprint arXiv:2409.15520* (2024).

[140] I. Loshchilov and F. Hutter, *Decoupled weight decay regularization*, *arXiv preprint arXiv:1711.05101* (2017).

[141] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms*, *arXiv preprint arXiv:1708.07747* (2017).